spectrum

Release 0.8.0

Thomas Cokelaer

Jul 25, 2021

Contents

Spectrum: a Spectral Analysis Library in Python

coverage 97%

Citation Cokelaer et al., (2017). 'Spectrum': Spectral Analysis in Python. Journal of Open Source Software, 2(18), 348, doi:10.21105/joss.00348

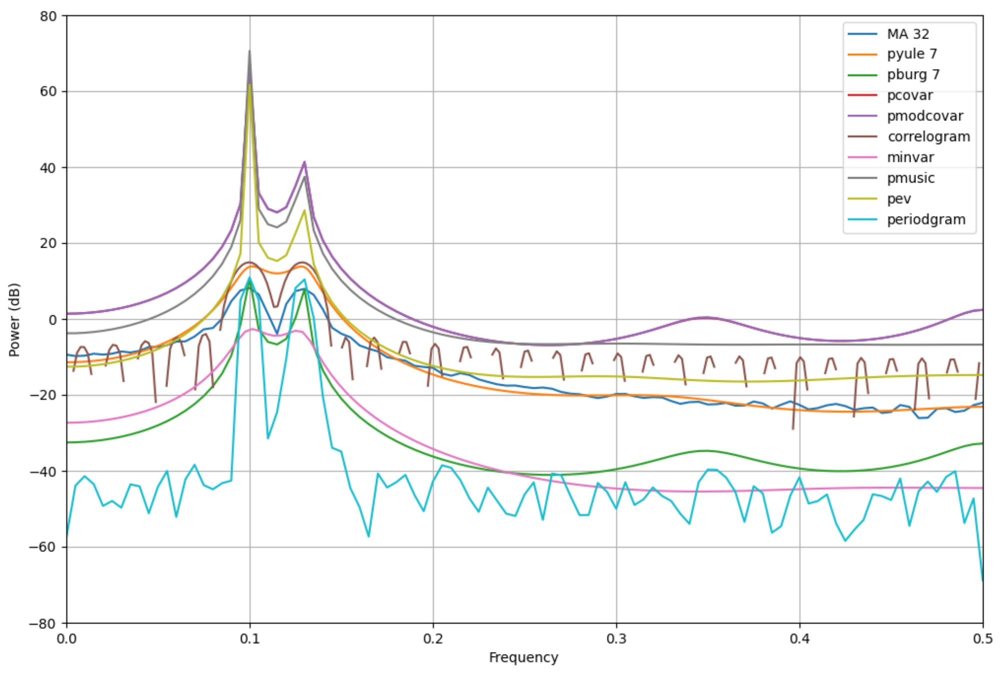
Contributions Please join https://github.com/cokelaer/spectrum

Contributors https://github.com/cokelaer/spectrum/graphs/contributors

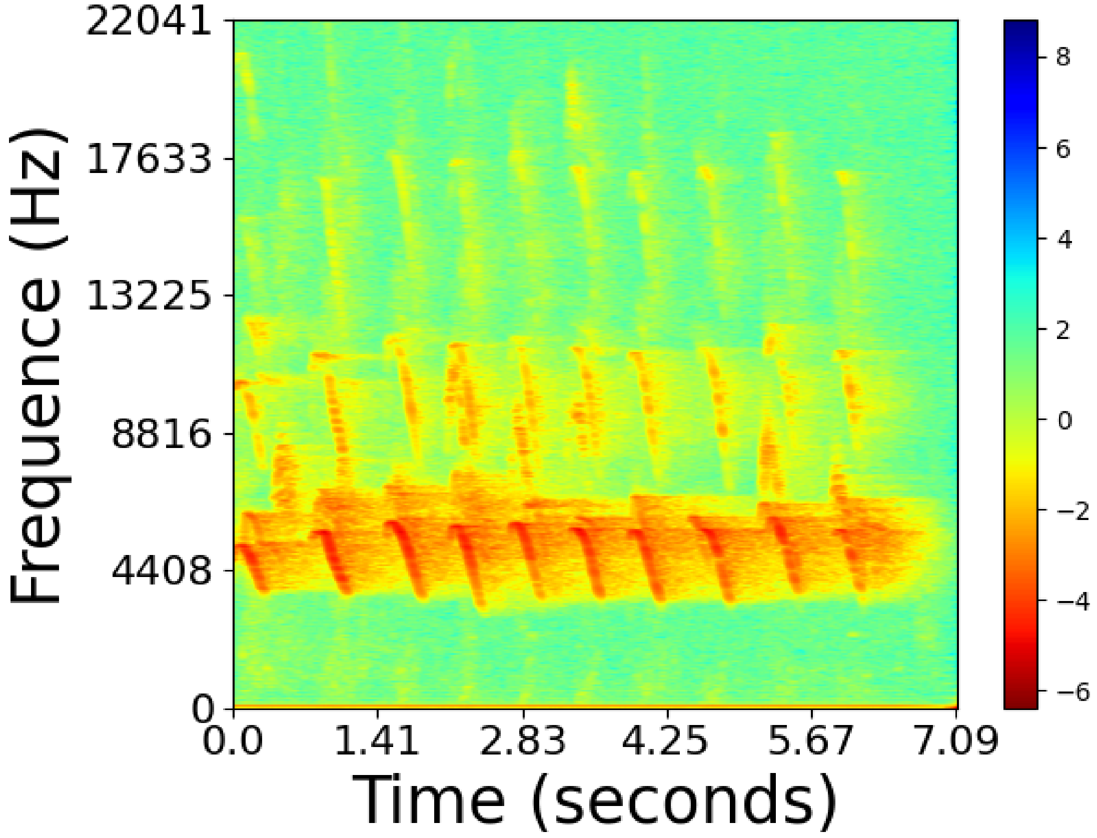
Issues Please use https://github.com/cokelaer/spectrum/issues

Documentation http://pyspectrum.readthedocs.io/

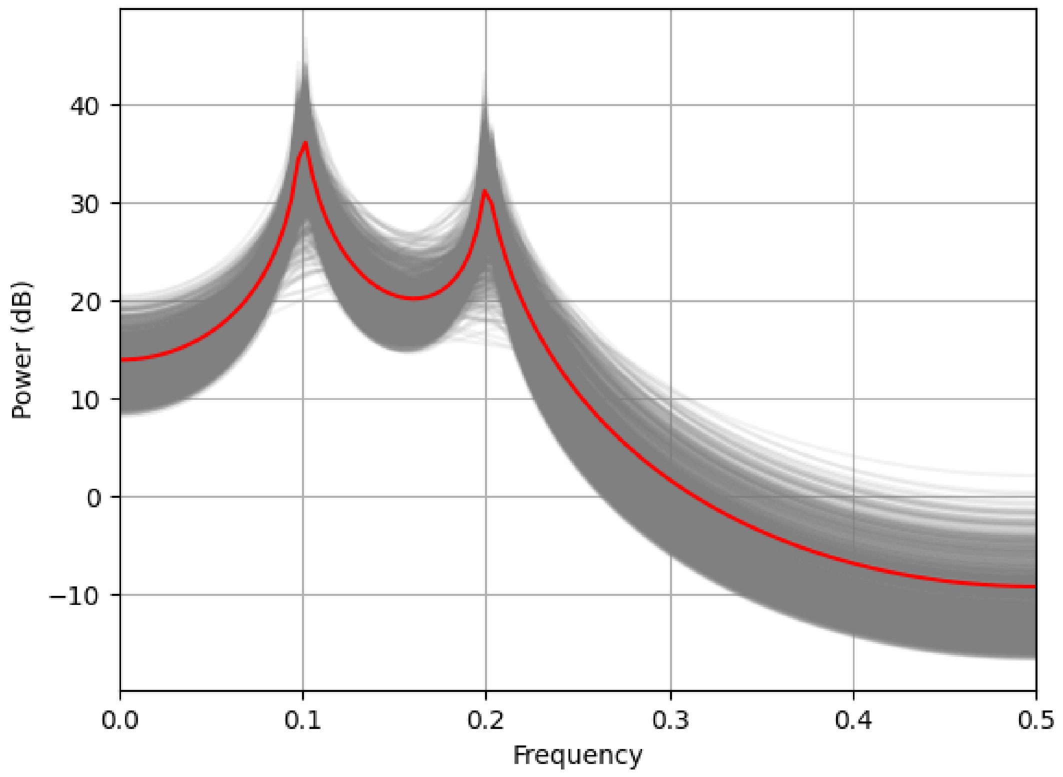
Spectrum is a Python library that contains tools to estimate Power Spectral Densities based on Fourier transform, Parametric methods or eigenvalues analysis. The Fourier methods are based upon correlogram, periodogram and Welch estimates. Standard tapering windows (Hann, Hamming, Blackman) and more exotic ones are available (DPSS, Taylor, . . .). The parametric methods are based on Yule-Walker, BURG, MA and ARMA, covariance and modified covariance methods. Non-parametric methods based on eigen analysis (e.g., MUSIC) and minimum variance analysis are also implemented. Finally, Multitapering combines several orthogonal tapering windows.



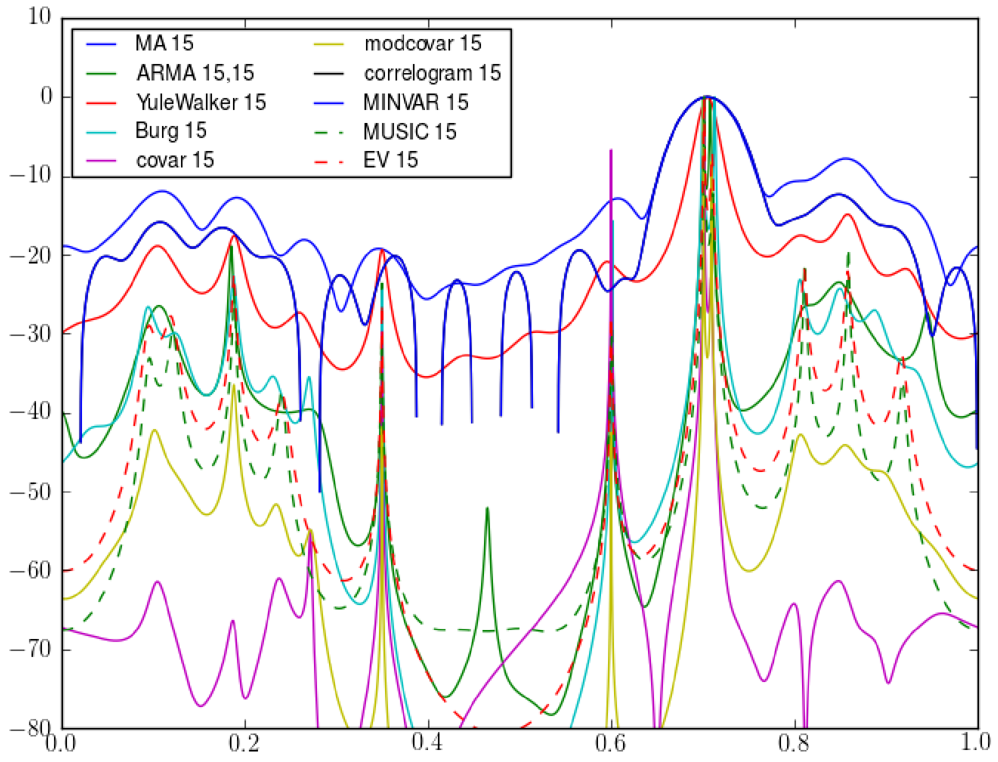
•


22041
17633
13225
8816
4408
0
Frequence (Hz)
0.0
1.41
2.83
4.25
5.67
7.09
Time (seconds)
8
6
4
2
0
−2
−4
−6

•


Power (dB)
40
30
20
10
0
−10
0.0
0.1
0.2
0.3
0.4
0.5
Frequency

•


MA 15
ARMA 15,15
YuleWalker 15
Burg 15
covar 15
modcovar 15
correlogram 15
MINVAR 15
MUSIC 15
EV 15
10
0
−10
−20
−30
−40
−50
−60
−70
−80
0.0
0.2
0.4
0.6
0.8
1.0

•

CHAPTER 1

Installation

1.1 Using pip

Spectrum is available on PYPi, so you should be able to type:

```
pip install spectrum
```

Since **spectrum** depends on other python packages such as Numpy, Matplotlib and Scipy they will be installed automatically (if not already installed).

You can also install the dependencies yourself by typing:

```
pip install numpy matplotlib scipy
```

Spectrum source code is available on Github https://github.com/cokelaer/spectrum

1.2 Conda installation

Spectrum is now available on CONDA. For Linux and MAC users, if you prefer to use conda, please use:

```
conda config --append channels conda-forge
conda install spectrum
```

1.3 From source and notes for developers

Developpers who want to get the source code can clone the repository:

```
git clone git@github.com:cokelaer/spectrum.git
cd spectrum
python setup.py install
```

Then, you can test the library using **pytest** or compile the documentation with Sphinx. To do so, install sphinx and other dependencies:

```
pip install --file requirements-dev.txt
```

CHAPTER 2

User Guide

Spectrum provides classes and functions to estimate Power Spectral Densities (PSD hereafter). This documentation will not describe PSD theoretical background, which can be found in many good books and references. Therefore, we consider that the reader is aware of some terminology used here below.

2.1 QuickStart (Periodogram example)

Spectrum can be invoked from a python shell. No GUI interface is provided yet. We recommend to use ipython, which should be started with the pylab option:

```
ipython --pylab
```

Then, you can import tools from **Spectrum** as follows:

```
from spectrum import Periodogram, data_cosine
```

Here we import a tool to compute a periodogram, and a tool to create some data. Indeed, we will use *data_cosine()* to generate a toy data sets:

```
data = data_cosine(N=1024, A=0.1, sampling=1024, freq=200)
```

where *data* contains a cosine signal with a frequency of 200Hz buried in white noise (amplitude 0.1). The data has a length of N=1024 and the sampling is 1024Hz.

We can analyse this data using one of the Power Spectrum Estimation method provided in spectrum. All methods can be found as functions or classes. Although we strongly recommend to use the object oriented approach, the functional approach may also be useful. For now, we will use the object approach because it provides more robustness and additional tools as compared to the functional approach (e.g., plotting). So, let us create a simple periodogram:

```
p = Periodogram(data, sampling=1024)
```

Here, we have created an object Periodogram. No computation has been performed yet. To run the actual estimation, you can use either:

```
p()
```

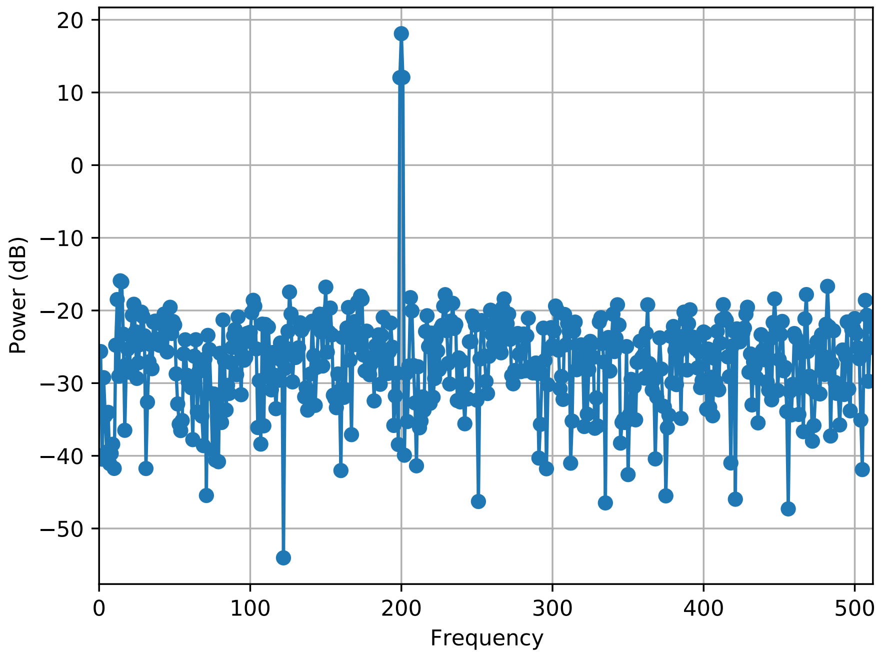
or:

```
p.run()
```

and finally, you can plot the resulting PSD:

```
p.plot(marker='o')  # standard matplotlib options are accepted
```

Warning: Changed in version 0.6.7: you do not need to use p() or p.run() anymore. It will be called automatically when using p.plot() or when you access to the *psd* attribute.



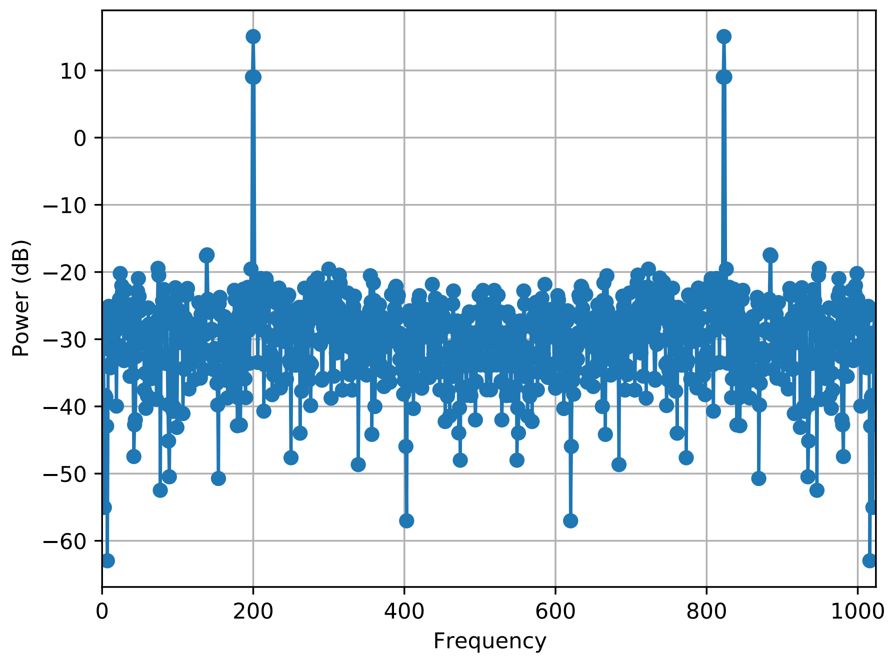
Since the data is purely real, the PSD (stored in p.psd) is a onesided PSD, with positive frequencies only. If the data were complex, the two-sided PSD would have been computed and plotted. For the real case, you can still plot a two-sided PSD by setting the sides option manually:

```
p.plot(sides='twosided')
```

You can also look at a centered PSD around the zero frequency:

```
p.plot(sides='centerdc')
```

Warning: By convention, the `psd` attribute contains the default PSD (either one-sided for real data or two-sided for the complex data).



Since **p** is an instance of Periodogram, you can introspect the object to obtain diverse information such as the original data, the sampling, the PSD itself and so on:

```
p.psd # contains the PSD values
p.frequencies() returns a list of frequencies
print(p) # prints some information about the PSD.
```

2.2 The object approach versus functional approach (ARMA example)

2.2.1 Object approach

In the previous section, we've already illustrated the object approach using a Fourier-based method with the simple periodogram method. In addition to the Fourier-based PSD estimates, **Spectrum** also provides parametric-based estimates. Let us use *parma()* class as a second illustrative example of the object approach:

```
from spectrum import parma
```

Many functionalities available in **Spectrum** are inspired by methods found in [Marple]. The data sample used in most of the examples is also taken from this reference and can be imported as follows (this is a 64 complex data samples):

```
from spectrum import marple_data
```

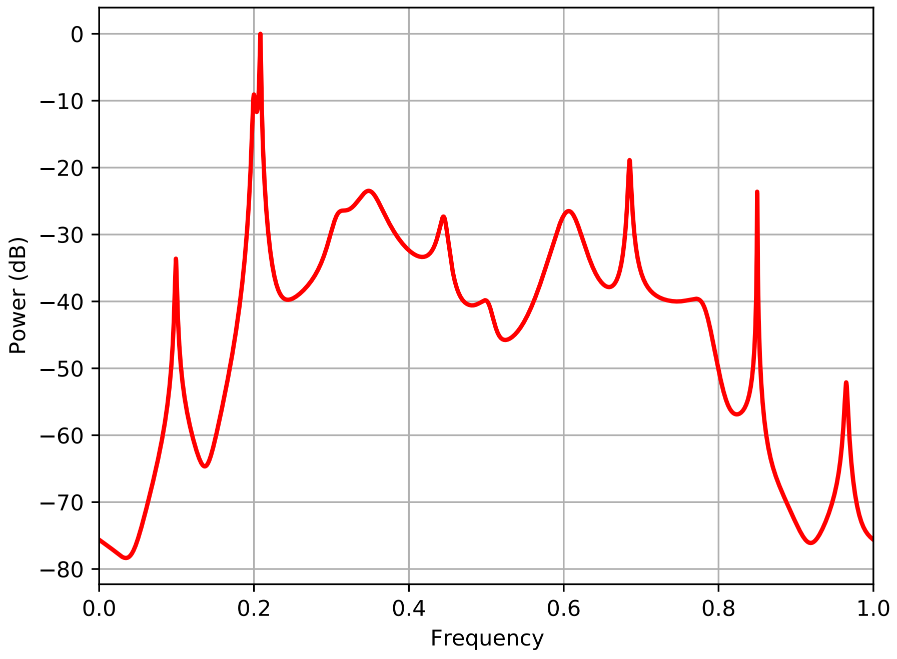
The class *parma* allows to create an ARMA model and to plot the PSD, similarly to the previous example (Periodogram). First, we need to create the object:

```
p = parma(marple_data, 15, 15, 30, NFFT=4096)
```

where 15,15 and 30 are arguments of the ARMA model (see `spectrum.parma`), and NFFT is the number of final points.

Then, computation and plot can be performed:

```
p.plot(norm=True, color='red', linewidth=2)
```



Since the data is complex, the PSD (stored in p.psd) is a twosided PSD. Note also that all optional arguments accepted by matplotlib function are also available in this implementation.

2.2.2 Functional approach

The object-oriented approach can be replaced by a functional one if required. Nevertheless, as mentionned earlier, this approach required more expertise and could easily lead to errors. The following example is identical to the previous piece of code.

In order to extract the autoregressive coefficients (AR) and Moving average coefficients (MA), the *`arma_estimate()`* can be used:

```
from spectrum.arma import arma_estimate, arma2psd
ar, ma, rho = arma_estimate(marple_data, 15, 15, 30)
```

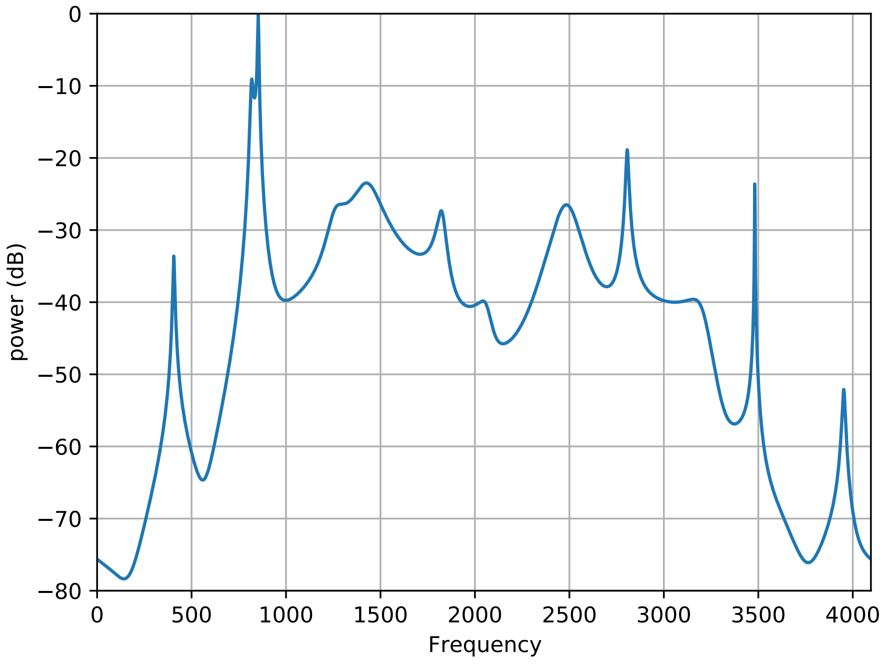
Once the AR and/or MA parameters are found, the *`arma2psd()`* function creates a two-sided PSD for you and the PSD can be plotted as follows:

```
from spectrum import arma_estimate, arma2psd, marple_data
from pylab import plot, axis, xlabel, ylabel, grid, log10
```

(continues on next page)

(continued from previous page)

```
ar, ma, rho = arma_estimate(marple_data, 15, 15, 30)
psd = arma2psd(ar, ma, rho=rho, NFFT=4096)
plot(10*log10(psd/max(psd)))
axis([0, 4096, -80, 0])
xlabel('Frequency')
ylabel('power (dB)')
grid(True)
```



Note:

1. The parameter 30 (line 3) is the correlation lag that should be twice as much as the required AR and MA coefficient number (see reference guide for details).
2. Then, we plot the PSD manually (line 5), and normalise it so as to use dB units (10*log10)
3. Since the data are complex data, the default plot is a two-sided PSD.
4. The frequency vector is not provided.

CHAPTER 3

Quick overview of spectral analysis methods

This section gives you a quick overview of the spectral analysis methods and classes that are available in **spectrum**. You will find the different classes associated to each PSD estimates. A functional approach is also possible but is not described here. See the reference guide for more details.

3.1 Non-parametric classes

The Fourier-based methods provides *`Periodogram`*, *`pcorrelogram`*, Welch estimate (not implemented see pylab.psd instead) and multitapering *`pmtm`*.

In addition to the Fourier-based methods, there are 3 types of non-parametric methods:

1. The Minimum of variance MV (Capon) is implemented in the class *`pminvar`*.
2. Two eigenvalues decomposition (MUSIC, eigenvalue) can be found in `pev` and *`pmusic`*.
3. Maximum entropy (MEM) (not yet implemented)

3.2 Autoregressive spectral estimation

There are essentially 3 methods to estimate the autoregressive (AR) parameters. The first one uses the autocorrelation sequence such as in the so-called **Yule-Walker** method (see *`pyule`*). A second method uses the reflection coefficient method such as in the **Burg** algorithm (see *`pburg`*). These methods minimise the forward prediction error (and backward) using Levinson recursion. Finally, a third important category of AR parameter method is based on the least squares linear prediction, which can be further decomposed into 2 categories. One that separate the minimization of the forward and backward linear prediction squared errors such as the **autocorrelation** or **covariance** methods (see *`pcovar`*). Another one that performs a combined minimization of the forward and backward prediction squared errors (**modified covariance**) (see *`pmodcovar`*).

Spectrum also provides *`parma`*, *`pma`* classes.

CHAPTER 4

Tutorials

Spectrum contains PSD estimates classes and methods but also many other functionalities that are useful in spectral estimation such as linear algebra tools (e.g., Levinson recursion), tapering windows, linear prediction and so on. This section provides several tutorials related to spectral estimation. They may require some expertise to fully understand them…

4.1 Yule Walker example

The following example illustrate the usage of the `aryule()` function that allows you to estimate the autoregressive coefficients of a set of data. First, we need some packages:

```
import scipy.signal
from spectrum import aryule
```

Then, we define a list of AR filter coefficients:

```
a = [1, -2.2137, 2.9403, -2.1697, 0.9606]
```

and create some noisy data with them:

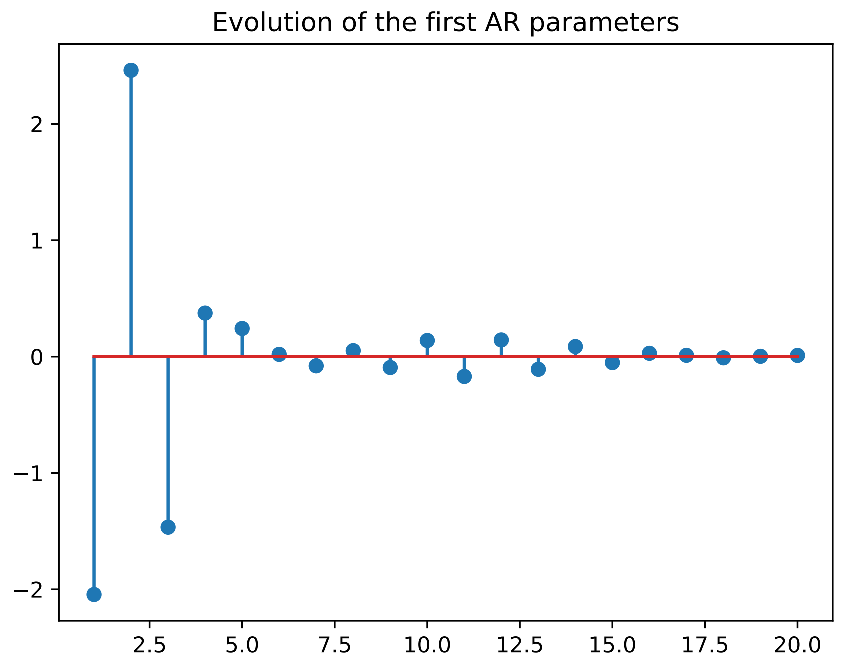
```
y = scipy.signal.lfilter([1], a, randn(1, 1024))
```

This array will be our data to test the Yule-Walker function, namely `aryule()`. Our goal is to estimate the AR coefficients from *y*. Since, we do not know the order of the autoregressive estimate, we first start by setting the order to 20:

```
ar, variance, coeff_reflection = aryule(y[0], 20)
```

By looking at the *coeff_reflection* output, it appears that the AR coefficient are rather small for order>4 (see following plot). From the plot, chosing an order 4 seems a reasonable choice.

It is possible to plot the PSD from the *ar* values using this:

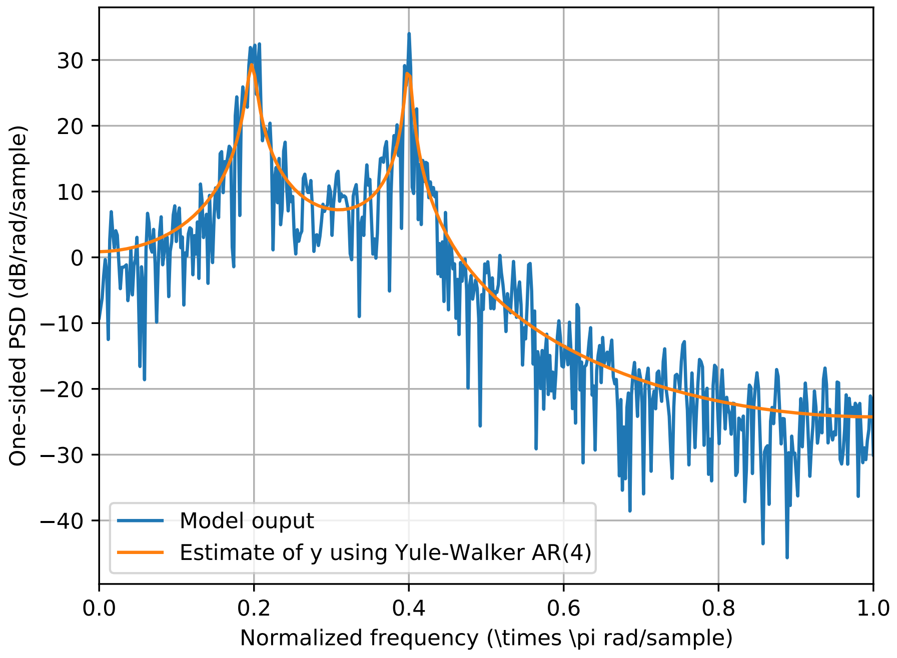


```
from pylab import log10, linspace, plot, xlabel, ylabel, legend, randn, pi
import scipy.signal
from spectrum import aryule, Periodogram, arma2psd
# Create a AR model
a = [1, -2.2137, 2.9403, -2.1697, 0.9606]
# create some data based on these AR parameters
y = scipy.signal.lfilter([1], a, randn(1, 1024))
# if we know only the data, we estimate the PSD using Periodogram
p = Periodogram(y[0], sampling=2)  # y is a list of list hence the y[0]
p.plot(label='Model ouput')

# now, let us try to estimate the original AR parameters
AR, P, k = aryule(y[0], 4)
PSD = arma2psd(AR, NFFT=512)
PSD = PSD[len(PSD):len(PSD)//2:-1]
plot(linspace(0, 1, len(PSD)), 10*log10(abs(PSD)*2./(2.*pi)),
    label='Estimate of y using Yule-Walker AR(4)')
xlabel(r'Normalized frequency (\times \pi rad/sample)')
ylabel('One-sided PSD (dB/rad/sample)')
legend()
```

This example uses the functional approach. Again, it is recommended to use the object approach with an instance from `pyule` (see quickstart section). The previous example would become even simpler:

```
from pylab import legend, randn
import scipy.signal
from spectrum import Periodogram, pyule
a = [1, -2.2137, 2.9403, -2.1697, 0.9606]
y = scipy.signal.lfilter([1], a, randn(1, 1024))
p = Periodogram(y[0], sampling=2)
```

(continues on next page)



(continued from previous page)

```
p.plot()
p = pyule(y[0], 4, sampling=2, scale_by_freq=False)
p.plot()
legend(['PSD of model output','PSD estimate of x using Yule-Walker AR(4)'])
```

4.2 PBURG example

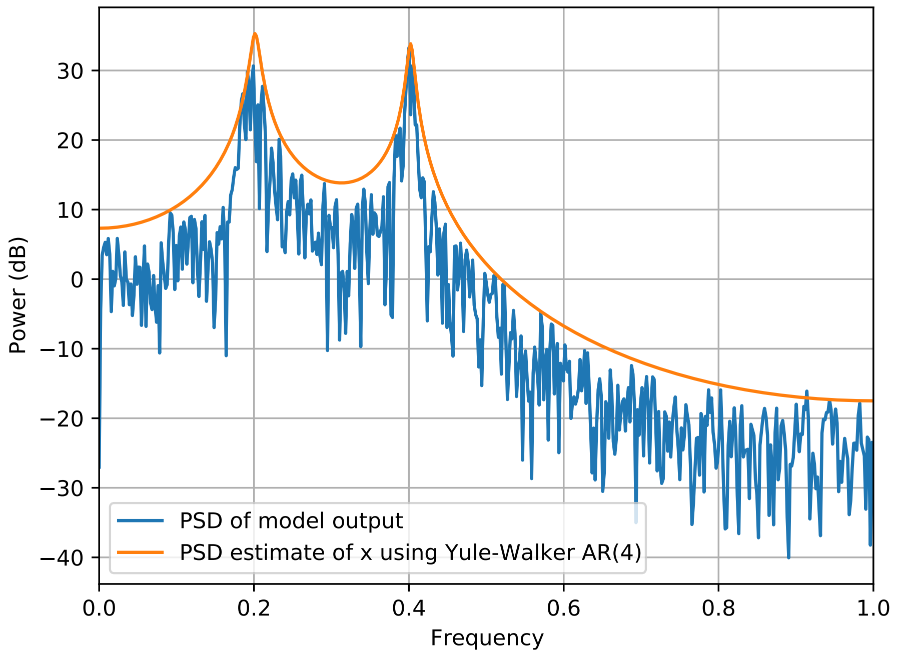
Here is another method to estimate an AR model, based on `arburg()` .

This example is inspired by an example found in Marple book. This is very similar to the previous example, where you will find more explanation (see yule-Walker tutorial).

```
from pylab import log10, pi, plot, xlabel, randn
import scipy.signal
from spectrum import arma2psd, arburg

# Define AR filter coefficients
a = [1, -2.2137, 2.9403, -2.1697, 0.9606];
```

```
[w,H] = scipy.signal.freqz(1, a, 256)
Hp = plot(w/pi, 20*log10(2*abs(H)/(2.*pi)),'r')
```

```
x = scipy.signal.lfilter([1], a, randn(256))
AR, rho, ref = arburg(x, 4)
```



```
PSD = arma2psd(AR, rho=rho, NFFT=512)
PSD = PSD[len(PSD):len(PSD)//2:-1]

plot(linspace(0, 1, len(PSD)), 10*log10(abs(PSD)*2./(2.*pi)))
xlabel('Normalized frequency (\times \pi rad/sample)')
```

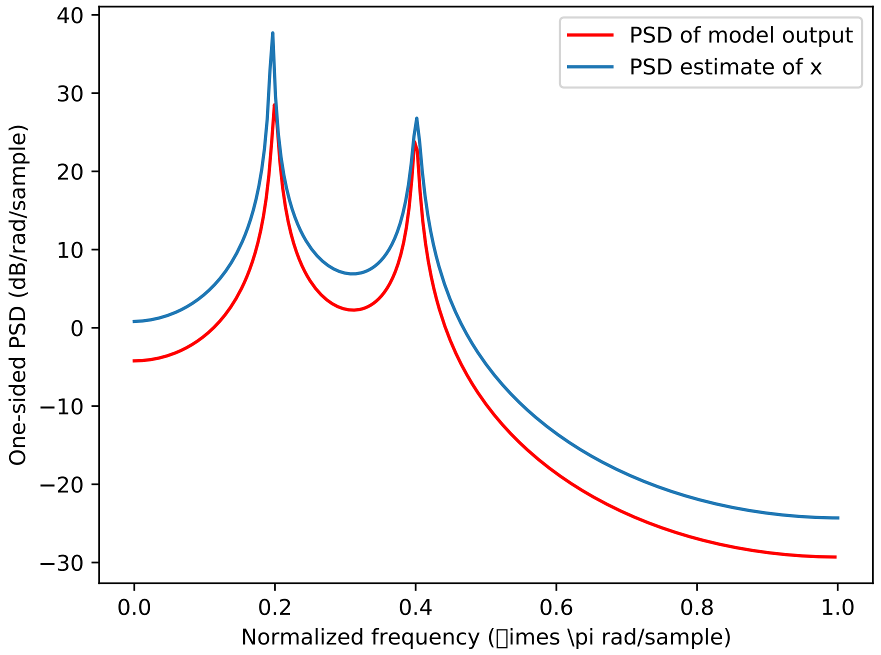
4.3 Variance outputs

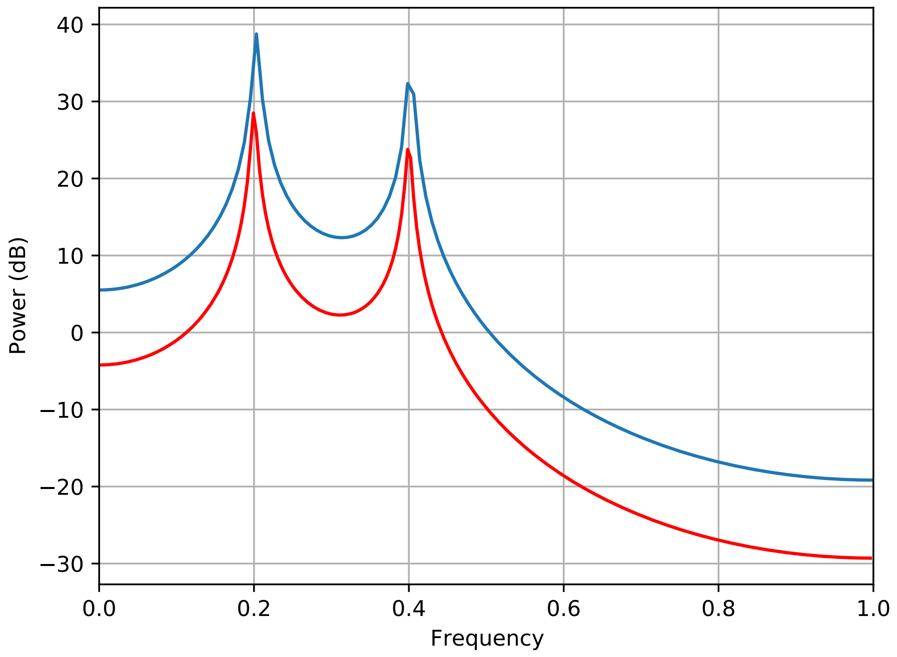
The `arburg()` function returns the AR parameters but also an estimation of the variance.

The following example plots the estimated variance (using arburg function) versus the true variance for different values of variance. In other words, we plot how accurately the variance can be estimated.

```
from pylab import plot, xlabel, ylabel, plot, axis, linspace, randn
import scipy.signal
from spectrum import arburg

# Define AR filter coefficients
a = [1, -2.2137, 2.9403, -2.1697, 0.9606];

# for different variance,
true_variance = linspace(0.1, 1, 20)
estimated_variance = []
for tv in true_variance:
    x = scipy.signal.lfilter([1], a, tv**0.5 * randn(1,256))
    AR, v, k = arburg(x[0], 4) # we estimate the AR parameter and variance
    estimated_variance.append(v)
```

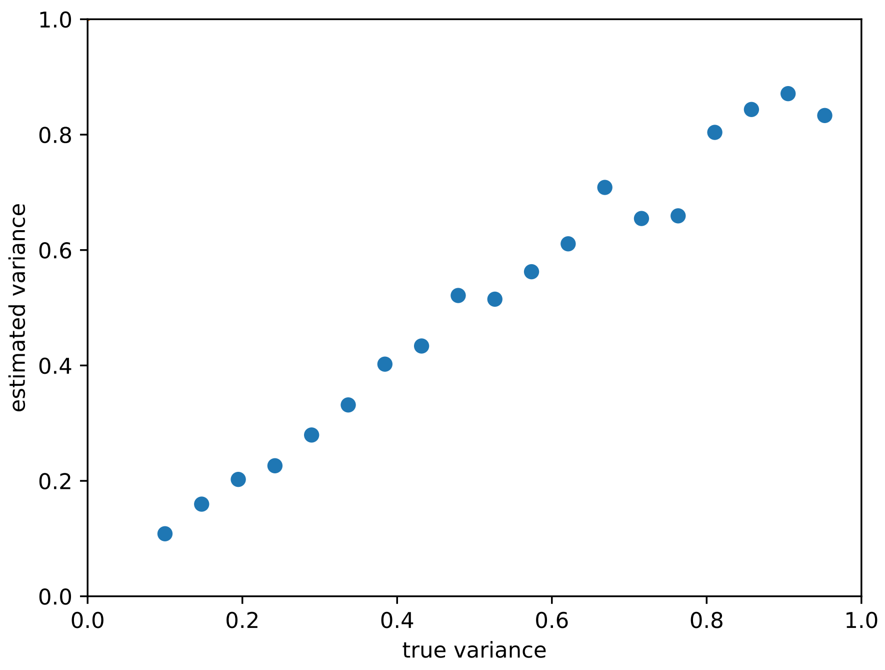
(continues on next page)


PSD of model output
PSD estimate of x
One-sided PSD (dB/rad/sample)
Normalized frequency (imes \pi rad/sample)
40
30
20
10
0
−10
−20
−30
0.0
0.2
0.4
0.6
0.8
1.0


Power (dB)
Frequency
40
30
20
10
0
−10
−20
−30
0.0
0.2
0.4
0.6
0.8
1.0

(continued from previous page)

```
plot(true_variance, estimated_variance, 'o')
xlabel('true variance')
ylabel('estimated variance')
plot([0,0],[1,1])
axis([0,1,0,1])
```



4.4 Windowing

Contents

In spectral analysis, it is common practice to multiply the input data by a tapering window.

Many windows are implemented and available in the `window` module as well as utilities to plot the window in time and frequency domains. Some windows that have been implemented are:

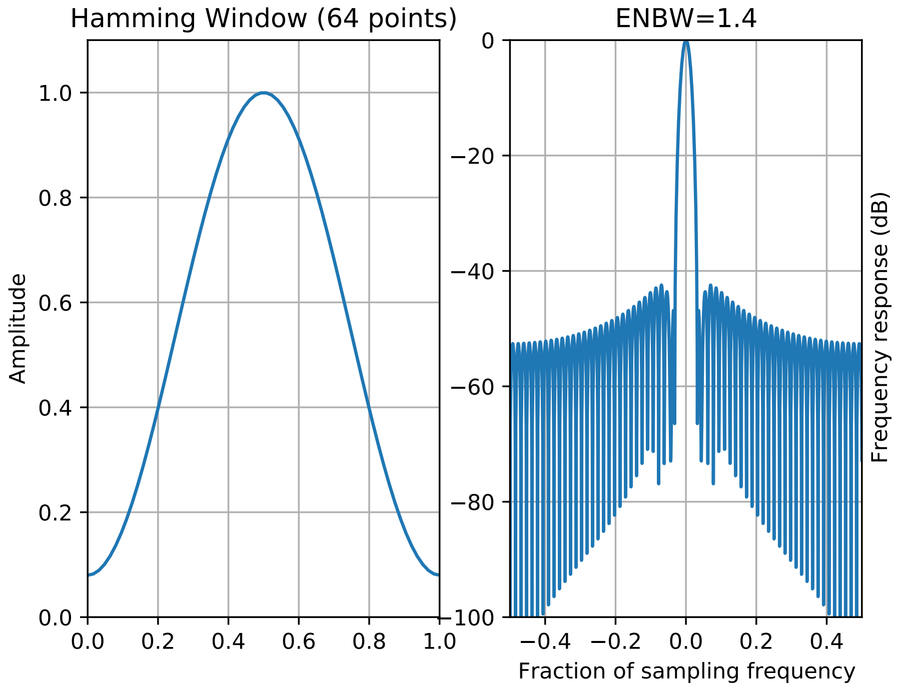
`spectrum.window.window_bartlett`(N)	Bartlett window (wrapping of numpy.bartlett) also known as Fejer
`spectrum.window.window_blackman`(N[, alpha])	Blackman window
`spectrum.window.window_gaussian`(N[, alpha])	Gaussian window
`spectrum.window.window_hamming`(N)	Hamming window
`spectrum.window.window_hann`(N)	Hann window (or Hanning).
`spectrum.window.window_kaiser`(N[, beta, method])	Kaiser window
`spectrum.window.window_lanczos`(N)	Lanczos window also known as sinc window.
`spectrum.window.window_nuttall`(N)	Nuttall tapering window
`spectrum.window.window_tukey`(N[, r])	Tukey tapering window (or cosine-tapered window)

See `window` module for a full list of windows. Note also that the `waveform` provides additional waveforms/windows.

4.4.1 Window object

There is a class `Window` that ease the manipulation of the tapering windows. It works as follows:

```
from spectrum.window import Window
N = 64
w = Window(N, 'hamming')
w.plot_time_freq()
```



where *N* is the length of the desired window, and “hamming” is the name. There are a lot of different windows, some of them require arguments. For example, the blackman window require an *alpha* argument:

```
w = Window(64, 'blackman', alpha=1)
```

From the object, you can easily access to the window data (*w.data*) and frequency (*w.frequencies*), as well as quantities such as the equivalent noise band width:

```
>>> from spectrum.window import Window
>>> N = 64
>>> w = Window(N, 'rectangular')
>>> w.enbw
1.0
```

To have a list of valid names, omit the name. It should raise an error with the list of valid names. Alternatively, type:

```
window_names.keys()
```

Finally, when a window require arguments, you need to know their names (e.g., in the blackman example above, the *alpha* parameter is required).

The only way to get this information is to look at the function *window_<name>* (e.g. window_blackman) and type:

```
window_blackman?
```

4.4.2 Simple window function call

You can explore the module to get the window function. For instance, if you look for the Hamming window, you should find a function called `window_hamming()`. You can look at it as follows:

```
from spectrum.window import window_hamming
from pylab import plot

N = 64
w = window_hamming(N)
plot(w)
```

4.4.3 Window Visualisation

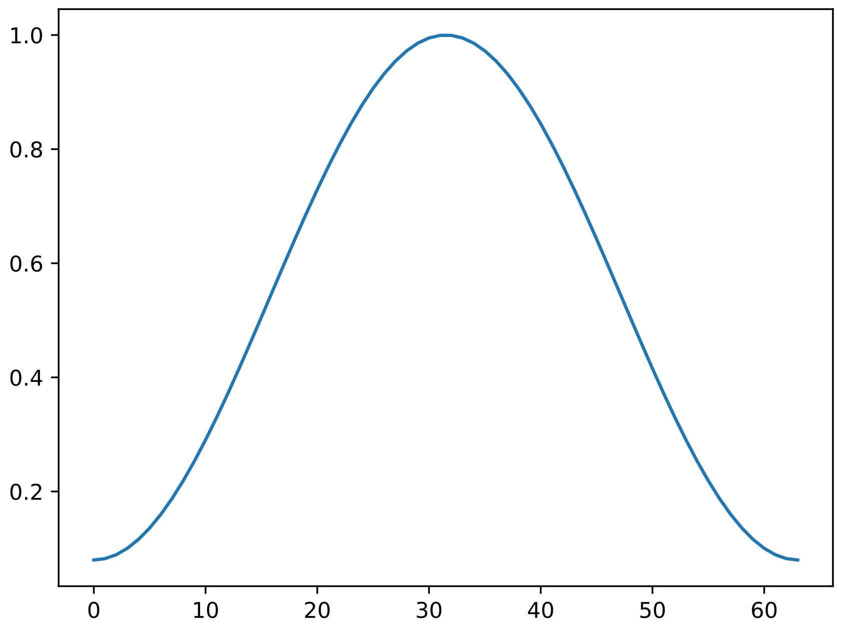
If you want to have a quick look at the window shape and its frequency behaviour, you can use the `window_visu()`:

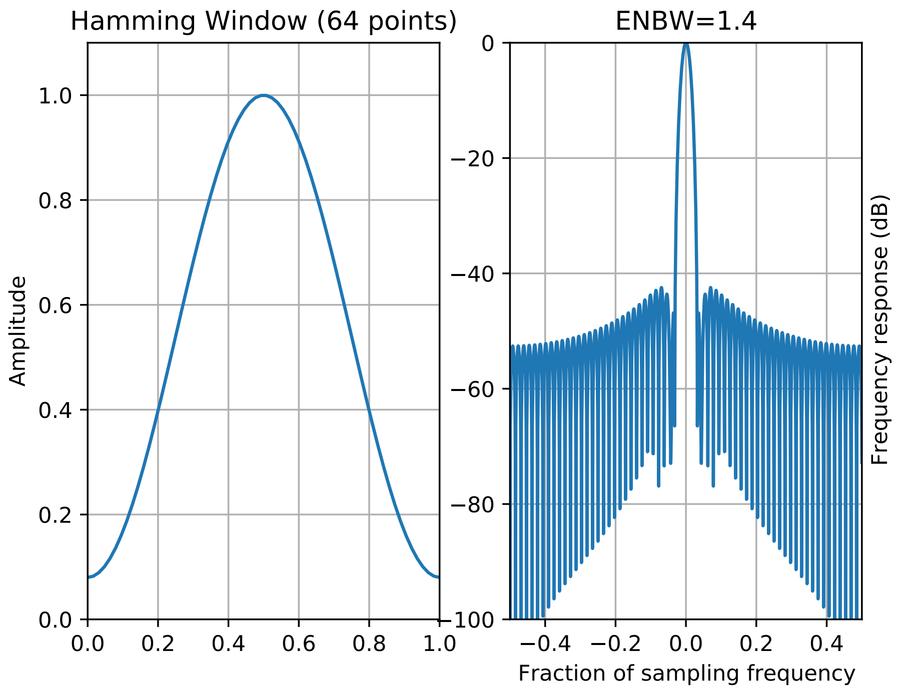
```
from spectrum.window import window_visu
N = 64
window_visu(N, 'hamming')
```

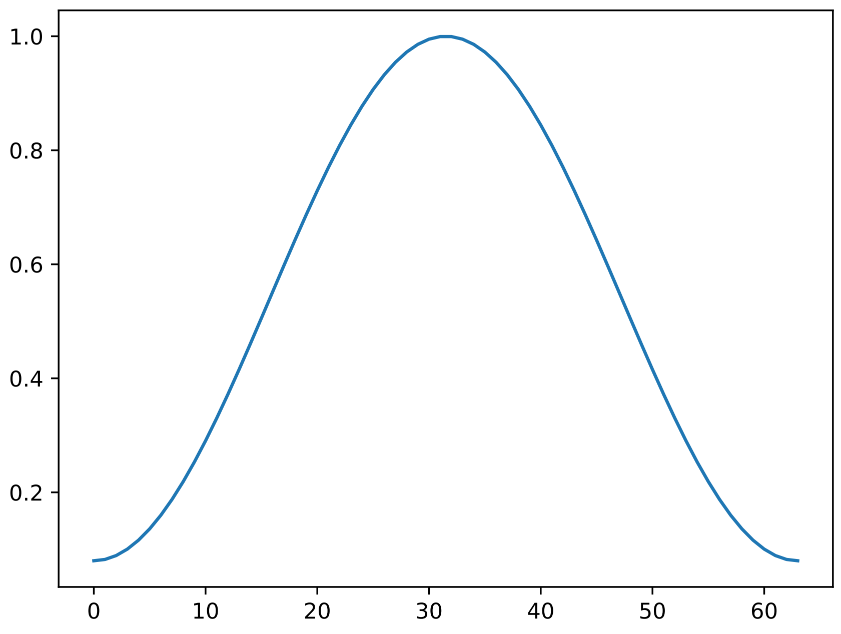
4.4.4 Window Factory

If you do not want the object approach, you may want to use the Factory function called `create_window()` (the `Window` class relies on this function). The previous Hamming window can be called using:

```
from spectrum.window import create_window
from pylab import plot

N = 64
w = create_window(N, 'hamming')
plot(w)
```


1.0
0.8
0.6
0.4
0.2
0
10
20
30
40
50
60


Hamming Window (64 points)
ENBW=1.4
Amplitude
Frequency response (dB)
Fraction of sampling frequency
0.0
0.2
0.4
0.6
0.8
1.0
0
−20
−40
−60
−80
−100
−0.4
−0.2
0.0
0.2
0.4



4.5 All PSD methods

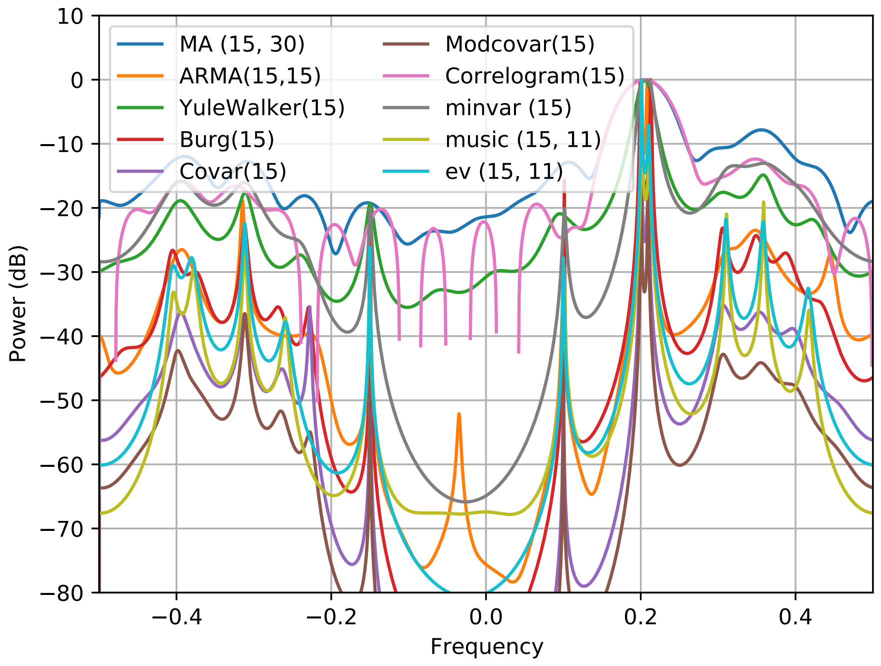
This example is used to generate the front image. It shows how to use the different PSD classes that can be found in **Spectrum**.

```
import spectrum
from spectrum.datasets import marple_data
from pylab import legend, ylim
norm = True
sides = 'centerdc'

# MA method
p = spectrum.pma(marple_data, 15, 30, NFFT=4096)
p(); p.plot(label='MA (15, 30)', norm=norm, sides=sides)

# ARMA method
p = spectrum.parma(marple_data, 15, 15, 30, NFFT=4096)
p(); p.plot(label='ARMA(15,15)', norm=norm, sides=sides)

# yulewalker
p = spectrum.pyule(marple_data, 15, norm='biased', NFFT=4096)
p(); p.plot(label='YuleWalker(15)', norm=norm, sides=sides)

#burg method
p = spectrum.pburg(marple_data, order=15, NFFT=4096)
p(); p.plot(label='Burg(15)', norm=norm, sides=sides)

#covar method
p = spectrum.pcovar(marple_data, 15, NFFT=4096)
p(); p.plot(label='Covar(15)', norm=norm, sides=sides)
```

(continues on next page)

(continued from previous page)

```
#modcovar method
p = spectrum.pmodcovar(marple_data, 15, NFFT=4096)
p(); p.plot(label='Modcovar(15)', norm=norm, sides=sides)

# correlagram
p = spectrum.pcorrelogram(marple_data, lag=15, NFFT=4096)
p(); p.plot(label='Correlogram(15)', norm=norm, sides=sides)

#minvar
p = spectrum.pminvar(marple_data, 15, NFFT=4096)
p(); p.plot(label='minvar (15)', norm=norm, sides=sides)

#music
p = spectrum.pmusic(marple_data, 15, 11, NFFT=4096)
p(); p.plot(label='music (15, 11)', norm=norm, sides=sides)

#ev
p = spectrum.pev(marple_data, 15, 11, NFFT=4096)
p(); p.plot(label='ev (15, 11)', norm=norm, sides=sides)

legend(loc='upper left', prop={'size':10}, ncol=2)
ylim([-80,10])
```



4.6 What is the Spectrum object ?

Normally Users should not be bother by the classes used. For instance if you use the pburg class to compute a PSD estimate base on the Burg method, you just nee to use `pburg`. Indeed, the normal usage to estimate a PSD is to use the PSD estimate starting with the letter *p* such as parma, pminvar, pburg, (exception: use Periodogram instead of pPeriodogram).

Yet, it may be useful for some advanced users and developers to know that all PSD estimates are based upon the `Spectrum` class (used by specialised classes such as `FourierSpectrum` and `ParametricSpectrum`).

The following example shows how to use `Spectrum`. First, let us create a Spectrum instance (first argument is the time series/data):

```
from spectrum import Spectrum, data_cosine, speriodogram, minvar
p = Spectrum(data_cosine(), sampling=1024)
```

Some information are stored and can be retrieved later on:

```
p.N
p.sampling
```

However, for now it contains no information about the PSD estimation method. For instance, if you type:

```
p.psd
```

it should return a warning message telling you that the PSD has not yet been computed. You can compute it either independantly, and set the *psd* attribute manually:

```
psd = speriodogram(p.data)
```

or you can associate a function to the *method* attribute:

```
p.method = minvar
```

and then call the function with the proper optional arguments:

```
p(15, NFFT=4096)
```

In both cases, the PSD is now saved in the *psd* attribute.

Of course, if you already know the method you want to use, then it is much simpler to call the appropriate class directly as shown in previous sections and examples:

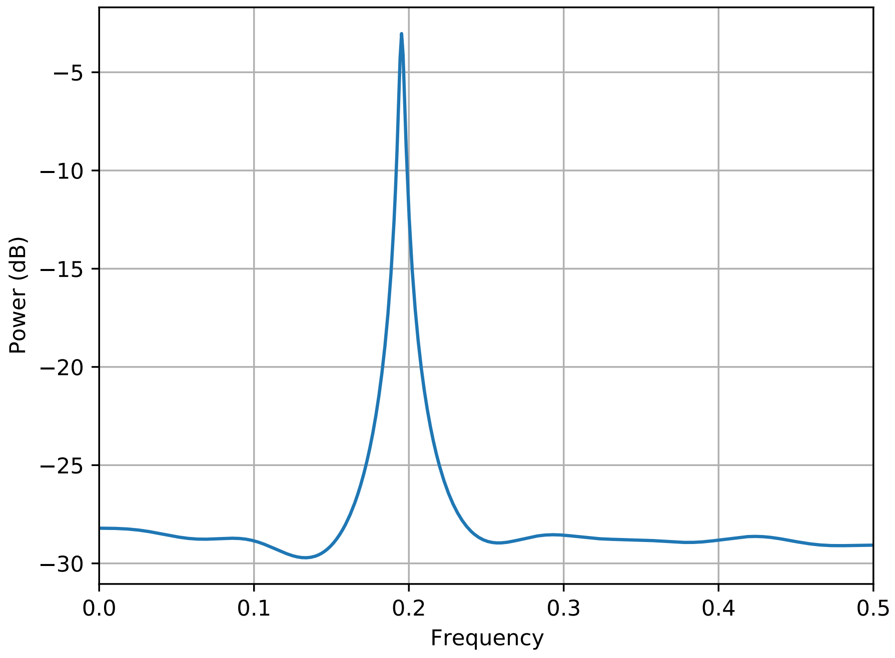
```
p =  pminvar(data_cosine(), 15)
p()
p.plot()
```

4.7 Criteria for Parametric methods

In order to estimate the order of a parametric model, one chose a PSD method such as the `aryule()` function. This function (when given an order) returns a list of AR parameters. The order selected may not be optimal (too low or too high). One tricky question is then to find a criteria to select this order in an optiaml way. Criteria are available and the following example illustrate their usage.

4.7.1 Example 1

Let us consider a data set (the Marple data already used earlier). We use the aryule function to estimate the AR parameter. This function also returns a parameter called *rho*. This parameter together with the length of the data and the selected order can be used by criteria functions such as the `AIC()` function to figure out the optimal order.



```
import spectrum
from spectrum.datasets import marple_data
import pylab

order = pylab.arange(1, 25)
rho = [spectrum.aryule(marple_data, i, norm='biased')[1] for i in order]
pylab.plot(order, spectrum.AIC(len(marple_data), rho, order), label='AIC')
```

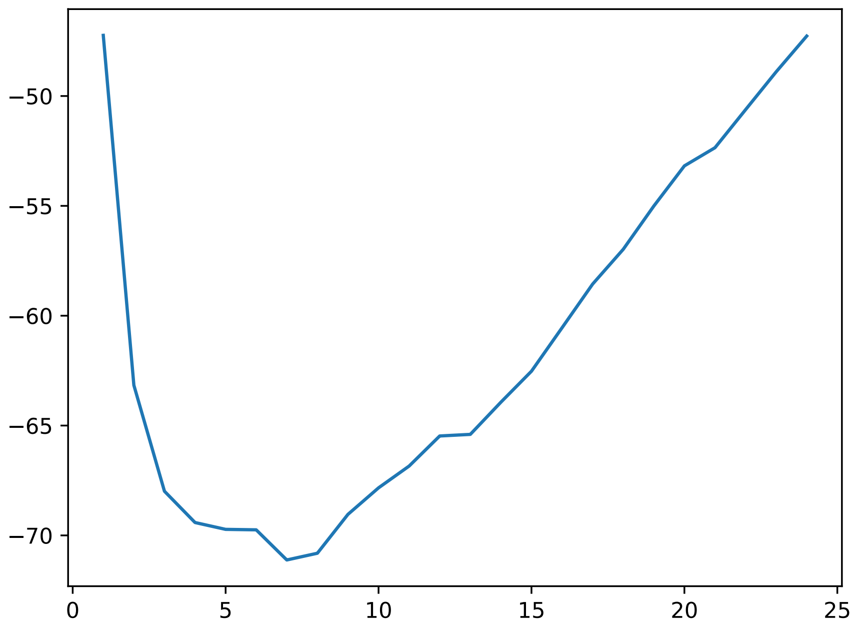
The optimal order corresponds to the minimal of the plotted function.

4.7.2 Example 2

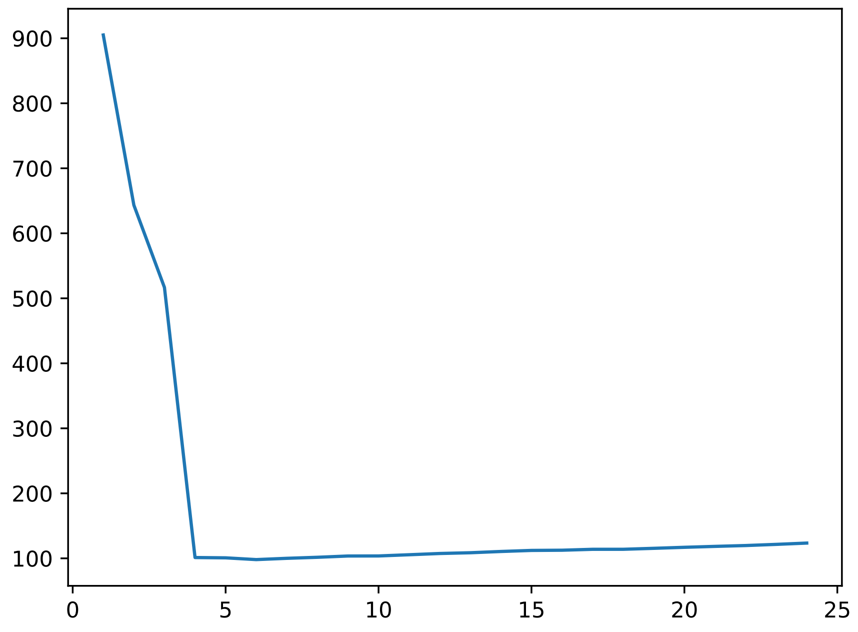
We can look at another example that was look at earlier with a AR(4):

```
import spectrum
from spectrum.datasets import marple_data
import scipy.signal
import pylab

# Define AR filter coefficients and some data accordingly
a = [1, -2.2137, 2.9403, -2.1697, 0.9606];
x = scipy.signal.lfilter([1], a, pylab.randn(1,256))

# study different order
order = pylab.arange(1, 25)
rho = [spectrum.aryule(x[0], i, norm='biased')[1] for i in order]
pylab.plot(order, spectrum.AIC(len(x[0]), rho, order), label='AIC')
```

Here, is appears that an order of 4 (at least) should be used, which correspond indeed to the original choice.



−50
−55
−60
−65
−70
0
5
10
15
20
25



900
800
700
600
500
400
300
200
100
0
5
10
15
20
25

CHAPTER 5

Reference guide

5.1 Fourier Methods

5.1.1 Power Spectrum Density based on Fourier Spectrum

`default_NFFT = 4096`
default number of samples used to compute FFT

`randn`(*d0*, *d1*, *...*, *dn*)
Return a sample (or samples) from the "standard normal" distribution.

Note: This is a convenience function for users porting code from Matlab, and wraps *standard_normal*. That function takes a tuple to specify the size of the output, which is consistent with other NumPy functions like *numpy.zeros* and *numpy.ones*.

Note: New code should use the `standard_normal` method of a `default_rng()` instance instead; please see the random-quick-start.

If positive int_like arguments are provided, *randn* generates an array of shape `(d0, d1, ..., dn)`, filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1. A single float randomly sampled from the distribution is returned if no argument is provided.

Parameters

d0, d1, . . . , dn [int, optional] The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

Returns

Z [ndarray or float] A `(d0, d1, ..., dn)`-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

See also:

`standard_normal` Similar, but takes a tuple as its argument.

`normal` Also accepts mu and sigma arguments.

`Generator.standard_normal` which should be used for new code.

Notes

For random samples from $N(\mu, \sigma^2)$, use:

`sigma * np.random.randn(...) + mu`

Examples

```
>>> np.random.randn()
2.1923875335537315  # random
```

Two-by-four array of samples from N(3, 6.25):

```
>>> 3 + 2.5 * np.random.randn(2, 4)
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```

`spectrum_set_level`(*level*)

This module provides Periodograms (classics, daniell, bartlett)

Periodogram(data[, sampling, window, NFFT, ...])	The Periodogram class provides an interface to periodogram PSDs
DaniellPeriodogram(data, P[, NFFT, detrend, ...])	Return Daniell's periodogram.
speriodogram(x[, NFFT, detrend, sampling, ...])	Simple periodogram, but matrices accepted.
WelchPeriodogram(data[, NFFT, sampling])	Simple periodogram wrapper of numpy.psd function.
speriodogram(x[, NFFT, detrend, sampling, ...])	Simple periodogram, but matrices accepted.

Code author: Thomas Cokelaer 2011

References See [Marple]

Usage

You can compute a periodogram using *speriodogram()*:

```
from spectrum import speriodogram, marple_data
from pylab import plot
p = speriodogram(marple_data)
plot(p)
```

However, the output is not always easy to manipulate or plot, therefore it is advised to use the class *Periodogram* instead:

```
from spectrum import Periodogram, marple_data
p = Periodogram(marple_data)
p.plot()
```

This class will take care of the plotting and internal state of the computation. For instance, if you can change the output easily:

```
p.plot(sides='twosided')
```

class pdaniell(*data*, *P*, *sampling=1.0*, *window='hann'*, *NFFT=None*, *scale_by_freq=True*, *detrend=None*)

The pdaniell class provides an interface to DaniellPeriodogram

```
from spectrum import data_cosine, pdaniell
data = data_cosine(N=4096, sampling=4096)
p = pdaniell(data, 8, NFFT=4096)
p.plot()
```

pdaniell Constructor

Parameters

- **data** (*array*) – input data (list or numpy.array)
- **P** (*int*) – number of neighbours to average over.
- **sampling** (*float*) – sampling frequency of the input `data`.
- **window** (*str*) – a tapering window. See *Window*.
- **NFFT** (*int*) – total length of the final data sets (padded with zero if needed; default is 4096)
- **scale_by_freq** (*bool*) –
- **detrend** (*str*) –

speriodogram(*x*, *NFFT=None*, *detrend=True*, *sampling=1.0*, *scale_by_freq=True*, *window='hamming'*, *axis=0*)

Simple periodogram, but matrices accepted.

Parameters

- **x** – an array or matrix of data samples.
- **NFFT** – length of the data before FFT is computed (zero padding)
- **detrend** (*bool*) – detrend the data before co,puteing the FFT
- **sampling** (*float*) – sampling frequency of the input `data`.
- **scale_by_freq** –
- **window** (*str*) –

Returns 2-sided PSD if complex data, 1-sided if real.

if a matrix is provided (using numpy.matrix), then a periodogram is computed for each row. The returned matrix has the same shape as the input matrix.

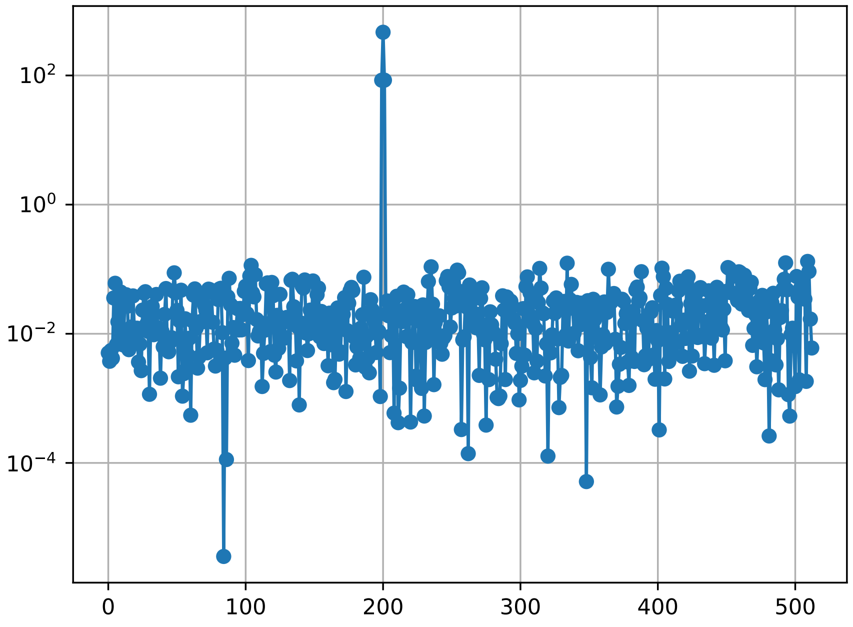
The mean of the input data is also removed from the data before computing the psd.

```
from pylab import grid, semilogy
from spectrum import data_cosine, speriodogram
data = data_cosine(N=1024, A=0.1, sampling=1024, freq=200)
```

(continues on next page)

(continued from previous page)

```
semilogy(speriodogram(data, detrend=False, sampling=1024), marker='o')
grid(True)
```



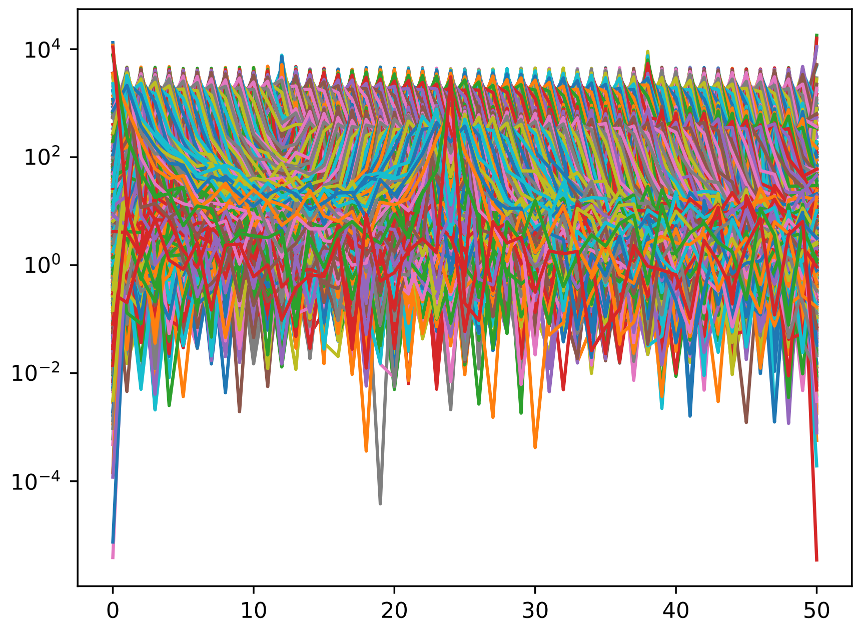
```
import numpy
from spectrum import speriodogram, data_cosine
from pylab import figure, semilogy, figure ,imshow
# create N data sets and make the frequency dependent on the time
N = 100
m = numpy.concatenate([data_cosine(N=1024, A=0.1, sampling=1024, freq=x)
    for x in range(1, N)]);
m.resize(N, 1024)
res = speriodogram(m)
figure(1)
semilogy(res)
figure(2)
imshow(res.transpose(), aspect='auto')
```

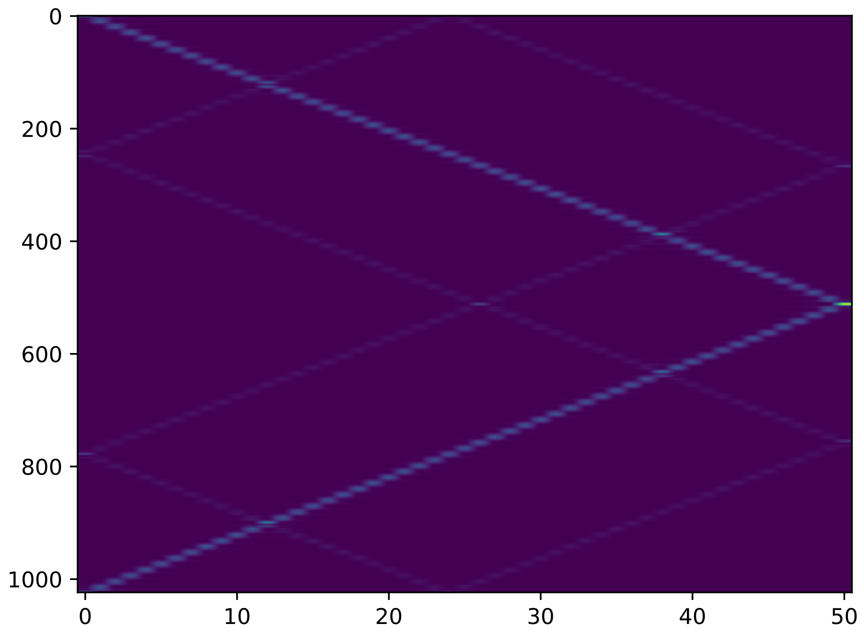
Todo: a proper spectrogram class/function that takes care of normalisation

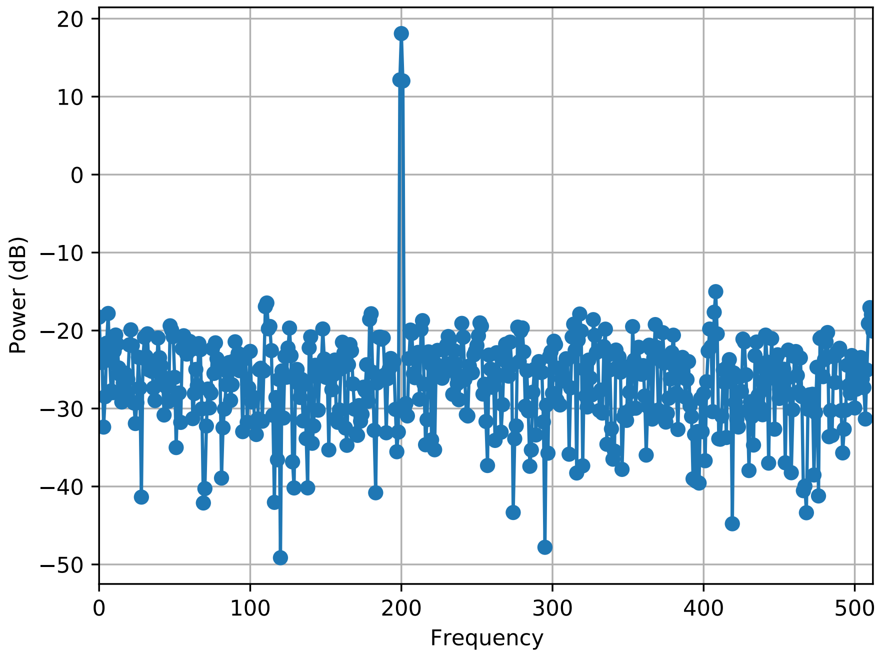
class Periodogram(*data*, *sampling=1.0*, *window='hann'*, *NFFT=None*, *scale_by_freq=False*, *detrend=None*)

The Periodogram class provides an interface to periodogram PSDs

```
from spectrum import Periodogram, data_cosine
data = data_cosine(N=1024, A=0.1, sampling=1024, freq=200)
p = Periodogram(data, sampling=1024)
p.plot(marker='o')
```


10^4
10^2
10^0
10^-2
10^-4
0
10
20
30
40
50


0
200
400
600
800
1000
0
10
20
30
40
50



Periodogram Constructor

Parameters

- **`data`** (*array*) – input data (list or numpy.array)
- **`sampling`** (*float*) – sampling frequency of the input `data`.
- **`window`** (*str*) – a tapering window. See *Window*.
- **`NFFT`** (*int*) – total length of the final data sets (padded with zero if needed; default is 4096)
- **`scale_by_freq`** (*bool*) –
- **`detrend`** (*str*) –

`WelchPeriodogram`(*data*, *NFFT=None*, *sampling=1.0*, ***kargs*)

Simple periodogram wrapper of numpy.psd function.

Parameters

- **`A`** – the input data
- **`NFFT`** (*int*) – total length of the final data sets (padded with zero if needed; default is 4096)
- **`window`** (*str*) –

Technical documentation

When we calculate the periodogram of a set of data we get an estimation of the spectral density. In fact as we use a Fourier transform and a truncated segments the spectrum is the convolution of the data with a rectangular window which Fourier transform is

$$W(s) = \frac{1}{N^2}\left[\frac{\sin(\pi s)}{\sin(\pi s/N)}\right]^2$$

Thus oscillations and sidelobes appears around the main frequency. One aim of t he tapering is to reduced this effects. We multiply data by a window whose sidelobes are much smaller than the main lobe. Classical window is hanning window. But other windows are available. However we must take into account this energy and divide the spectrum by energy of taper used. Thus periodogram becomes :

$$D_k \equiv \sum_{j=0}^{N-1} c_j w_j \; e^{2\pi ijk/N} \qquad k = 0, ..., N-1$$

$$P(0) = P(f_0) = \frac{1}{2\pi W_{ss}} |D_0|^2$$

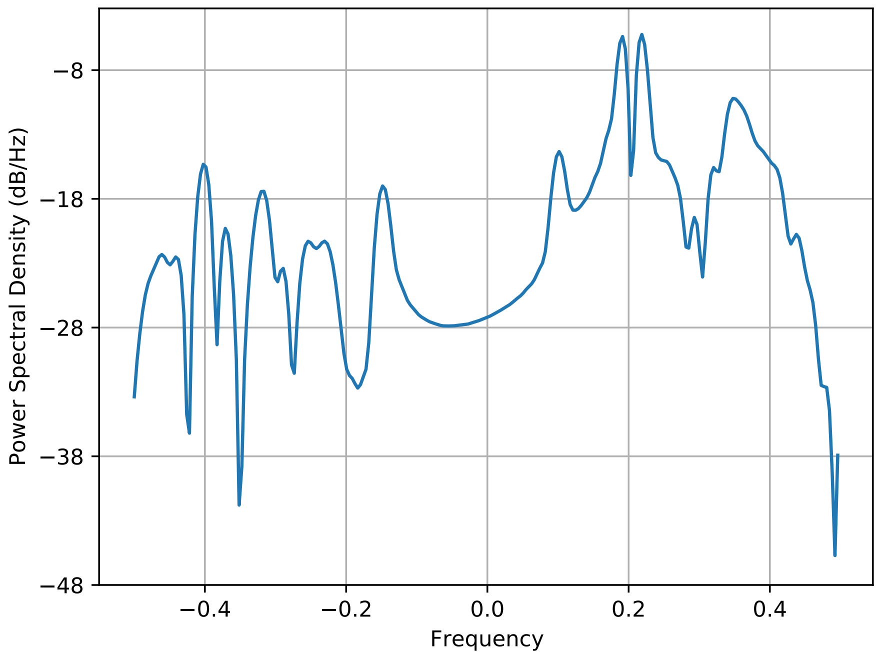
$$P(f_k) = \frac{1}{2\pi W_{ss}} \left[|D_k|^2 + |D_{N-k}|^2 \right] \qquad k = 0, 1, ..., \left(\frac{1}{2} - 1 \right)$$

$$P(f_c) = P(f_{N/2}) = \frac{1}{2\pi W_{ss}} |D_{N/2}|^2$$

with

$$W_{ss} \equiv N \sum_{j=0}^{N-1} w_j^2$$

```
from spectrum import WelchPeriodogram, marple_data
psd = WelchPeriodogram(marple_data, 256)
```



DaniellPeriodogram(*data*, *P*, *NFFT=None*, *detrend='mean'*, *sampling=1.0*, *scale_by_freq=True*, *window='hamming'*)

Return Daniell's periodogram.

To reduce fast fluctuations of the spectrum one idea proposed by daniell is to average each value with points in its neighboorhood. It's like a low filter.

$$\hat{P}_D[f_i] = \frac{1}{2P+1} \sum_{n=i-P}^{i+P} \tilde{P}_{xx}[f_n]$$

where P is the number of points to average.

Daniell's periodogram is the convolution of the spectrum with a low filter:

$$\hat{P}_D(f) = \hat{P}_{xx}(f) * H(f)$$

Example:

```
>>> DaniellPeriodogram(data, 8)
```

if N/P is not integer, the final values of the original PSD are not used.

using DaniellPeriodogram(data, 0) should give the original PSD.

Correlogram PSD estimates

This module provides Correlograms methods

`CORRELOGRAMPSD`(X[, Y, lag, window, norm, ...])	PSD estimate using correlogram method.
`pcorrelogram`(data[, sampling, lag, window, ...])	The Correlogram class provides an interface to `CORRELOGRAMPSD()`.

Code author: Thomas Cokelaer 2011

References See [Marple]

CORRELOGRAMPSD(*X*, *Y=None*, *lag=-1*, *window='hamming'*, *norm='unbiased'*, *NFFT=4096*, *window_params={}*, *correlation_method='xcorr'*)

PSD estimate using correlogram method.

Parameters

- **X** (*array*) – complex or real data samples X(1) to X(N)
- **Y** (*array*) – complex data samples Y(1) to Y(N). If provided, computes the cross PSD, otherwise the PSD is returned
- **lag** (*int*) – highest lag index to compute. Must be less than N
- **window_name** (*str*) – see `window` for list of valid names
- **norm** (*str*) – one of the valid normalisation of `xcorr()` (biased, unbiased, coeff, None)
- **NFFT** (*int*) – total length of the final data sets (padded with zero if needed; default is 4096)
- **correlation_method** (*str*) – either *xcorr* or *CORRELATION*. CORRELATION should be removed in the future.

Returns

- Array of real (cross) power spectral density estimate values. This is a two sided array with negative values following the positive ones whatever is the input data (real or complex).

Description:

The exact power spectral density is the Fourier transform of the autocorrelation sequence:

$$P_{xx}(f) = T \sum_{m=-\infty}^{\infty} r_{xx}[m] exp^{-j2\pi fmT}$$

The correlogram method of PSD estimation substitutes a finite sequence of autocorrelation estimates $\hat{r}_{xx}$ in place of r_{xx}. This estimation can be computed with `xcorr()` or `CORRELATION()` by chosing a proprer lag *L*. The estimated PSD is then

$$\hat{P}_{xx}(f) = T \sum_{m=-L}^{L} \hat{r}_{xx}[m] exp^{-j2\pi fmT}$$

The lag index must be less than the number of data samples *N*. Ideally, it should be around *L/10* [Marple] so as to avoid greater statistical variance associated with higher lags.

To reduce the leakage of the implicit rectangular window and therefore to reduce the bias in the estimate, a tapering window is normally used and lead to the so-called Blackman and Tukey correlogram:

$$\hat{P}_{BT}(f) = T \sum_{m=-L}^{L} w[m] \hat{r}_{xx}[m] exp^{-j2\pi fmT}$$

The correlogram for the cross power spectral estimate is

$$\hat{P}_{xx}(f) = T \sum_{m=-L}^{L} \hat{r}_{xx}[m] exp^{-j2\pi fmT}$$

which is computed if `Y` is not provide. In such case, $r_{yx} = r_{xy}$ so we compute the correlation only once.

```
from spectrum import CORRELOGRAMPSD, marple_data
from spectrum.tools import cshift
from pylab import log10, axis, grid, plot,linspace

psd = CORRELOGRAMPSD(marple_data, marple_data, lag=15)
f = linspace(-0.5, 0.5, len(psd))
psd = cshift(psd, len(psd)/2)
plot(f, 10*log10(psd/max(psd)))
axis([-0.5,0.5,-50,0])
grid(True)
```

See also:

`create_window()`, `CORRELATION()`, `xcorr()`, *`pcorrelogram`*.

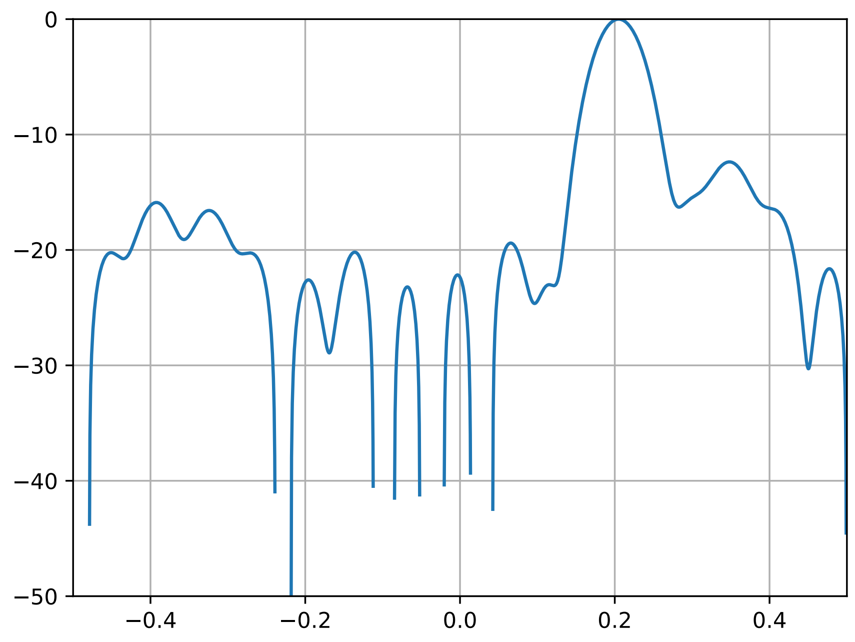
class pcorrelogram(*data*, *sampling=1.0*, *lag=-1*, *window='hamming'*, *NFFT=None*, *scale_by_freq=True*, *detrend=None*)

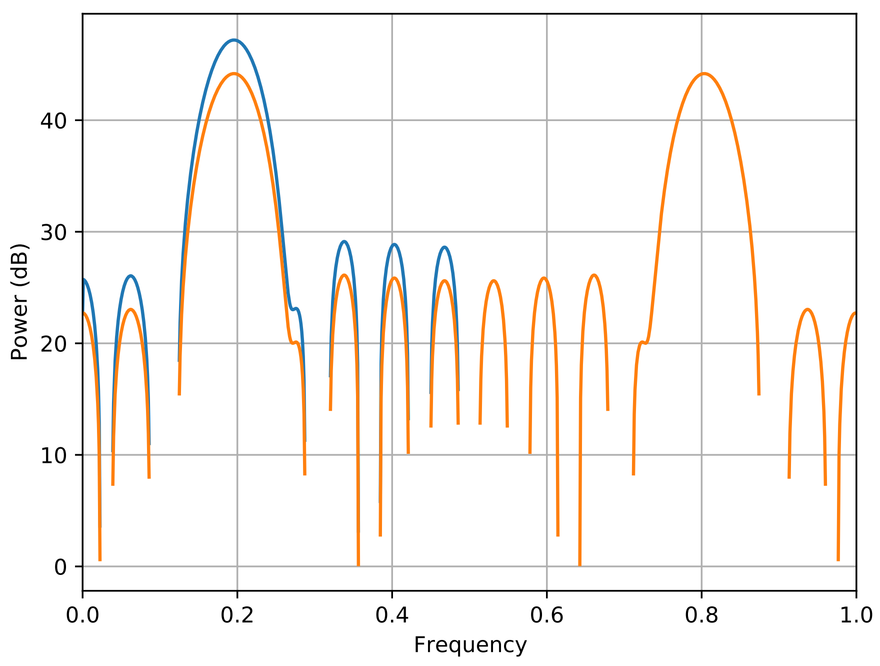
The Correlogram class provides an interface to *`CORRELOGRAMPSD()`*.

It returns an object that inherits from `FourierSpectrum` and therefore ease the manipulation of PSDs.

```
from spectrum import pcorrelogram, data_cosine
p = pcorrelogram(data_cosine(N=1024), lag=15)
p.plot()
p.plot(sides='twosided')
```

Correlogram Constructor


0
−10
−20
−30
−40
−50
−0.4
−0.2
0.0
0.2
0.4


40
30
20
10
0
Power (dB)
0.0
0.2
0.4
0.6
0.8
1.0
Frequency

Parameters

- **data** (*array*) – input data (list or numpy.array)
- **sampling** (*float*) – sampling frequency of the input `data`.
- **lag** (*int*) –
- **window** (*str*) – a tapering window. See `Window`.
- **NFFT** (*int*) – total length of the final data sets (padded with zero if needed; default is 4096)
- **scale_by_freq** (*bool*) –
- **detrend** (*str*) –

5.1.2 Tapering Windows

This module contains tapering windows utilities.

`Window`(N[, name, norm])	Window tapering object
`window_visu`([N, name])	A Window visualisation tool
`create_window`(N[, name])	Returns the N-point window given a valid name
`window_hann`(N)	Hann window (or Hanning).
`window_hamming`(N)	Hamming window
`window_bartlett`(N)	Bartlett window (wrapping of numpy.bartlett) also known as Fejer
`window_bartlett_hann`(N)	Bartlett-Hann window
`window_blackman`(N[, alpha])	Blackman window
`window_blackman_harris`(N)	Blackman Harris window
`window_blackman_nuttall`(N)	Blackman Nuttall window
`window_bohman`(N)	Bohman tapering window
`window_chebwin`(N[, attenuation])	Cheb window
`window_cosine`(N)	Cosine tapering window also known as sine window.
`window_flattop`(N[, mode, precision])	Flat-top tapering window
`window_gaussian`(N[, alpha])	Gaussian window
`window_hamming`(N)	Hamming window
`window_hann`(N)	Hann window (or Hanning).
`window_kaiser`(N[, beta, method])	Kaiser window
`window_lanczos`(N)	Lanczos window also known as sinc window.
`window_nuttall`(N)	Nuttall tapering window
`window_parzen`(N)	Parsen tapering window (also known as de la Valle-Poussin)
`window_taylor`(N[, nbar, sll])	Taylor tapering window
`window_tukey`(N[, r])	Tukey tapering window (or cosine-tapered window)

Code author: Thomas Cokelaer 2011

References See [Nuttall], [Marple], [Harris]

class Window (*N*, *name=None*, *norm=True*, ***kargs*)
Window tapering object

This class provides utilities to manipulate tapering windows. Plotting functions allows to visualise the time and frequency response. It is also possible to retrieve relevant quantities such as the equivalent noise band width.

The following examples illustrates the usage. First, we create the window by providing a name and a size:

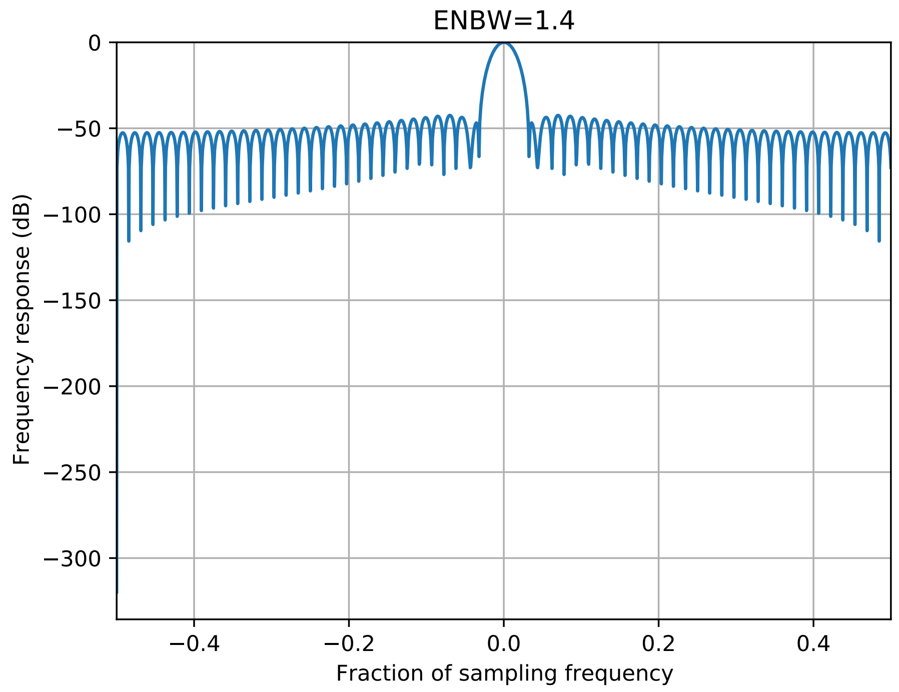
```
from spectrum import Window
w = Window(64, 'hamming')
```

The window has been computed and the data is stored in:

```
w.data
```

This object contains plotting methods so that you can see the time or frequency response.

```
from spectrum.window import Window
w = Window(64, 'hamming')
w.plot_frequencies()
```



Some windows may accept optional arguments. For instance, *window_blackman()* accepts an optional argument called α as well as *Window*. Indeed, we use the factory *create_window()*, which manage all the optional arguments. So you can write:

```
w = Window(64, 'blackman', alpha=1)
```

See also:

create_window().

Constructor:

Create a tapering window object

Parameters

- **N** – the window length
- **name** – the type of window, e.g., 'Hann'
- **norm** – normalise the window in frequency domain (for plotting)
- **kargs** – any of `create_window()` valid optional arguments.

Attributes:

- data: time series data
- frequencies: getter to the frequency series
- response: getter to the PSD
- enbw: getter to the Equivalent noise band width.

N
Getter for the window length

compute_response (***kargs*)
Compute the window data frequency response

Parameters

- **norm** – True by default. normalised the frequency data.
- **NFFT** (`int`) – total length of the final data sets(2048 by default. if less than data length, then NFFT is set to the data length*2).

The response is stored in `response`.

Note: Units are dB (20 log10) since we plot the frequency response)

data
Getter for the window values (in time)

enbw
getter for the equivalent noise band width. See `enbw()` function

frequencies
Getter for the frequency array

info ()
Print object information such as length and name

mean_square
returns :math:' rac{w^2}{N}'

name
Getter for the window name

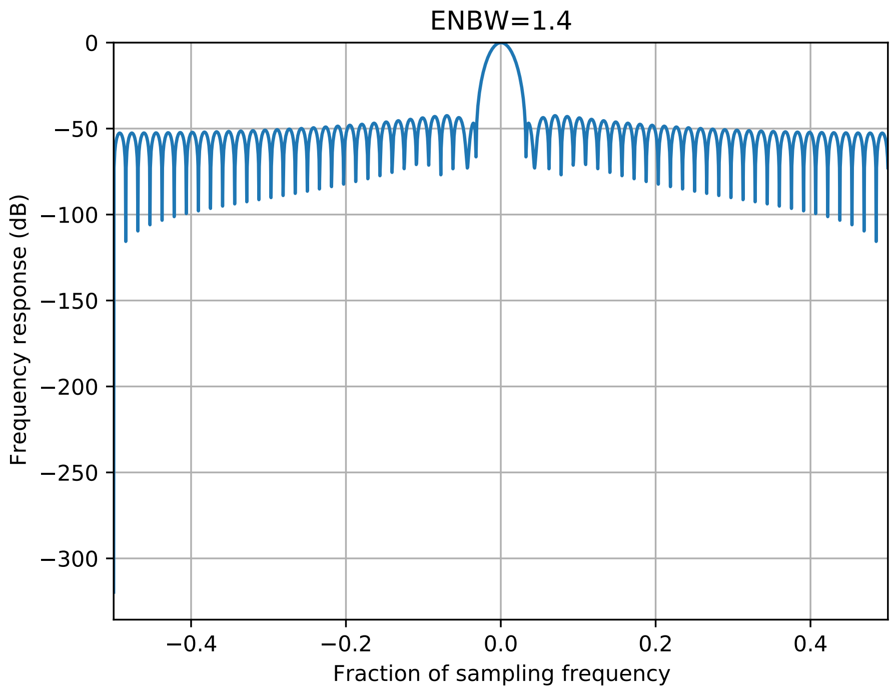
norm
Getter of the normalisation flag (True by default)

plot_frequencies (*mindB=None*, *maxdB=None*, *norm=True*)
Plot the window in the frequency domain

Parameters

- **mindB** – change the default lower y bound
- **maxdB** – change the default upper lower bound
- **norm** (*bool*) – if True, normalise the frequency response.

```
from spectrum.window import Window
w = Window(64, name='hamming')
w.plot_frequencies()
```



plot_time_freq(*mindB=-100*, *maxdB=None*, *norm=True*, *yaxis_label_position='right'*)
Plotting method to plot both time and frequency domain results.

See *plot_frequencies()* for the optional arguments.

```
from spectrum.window import Window
w = Window(64, name='hamming')
w.plot_time_freq()
```

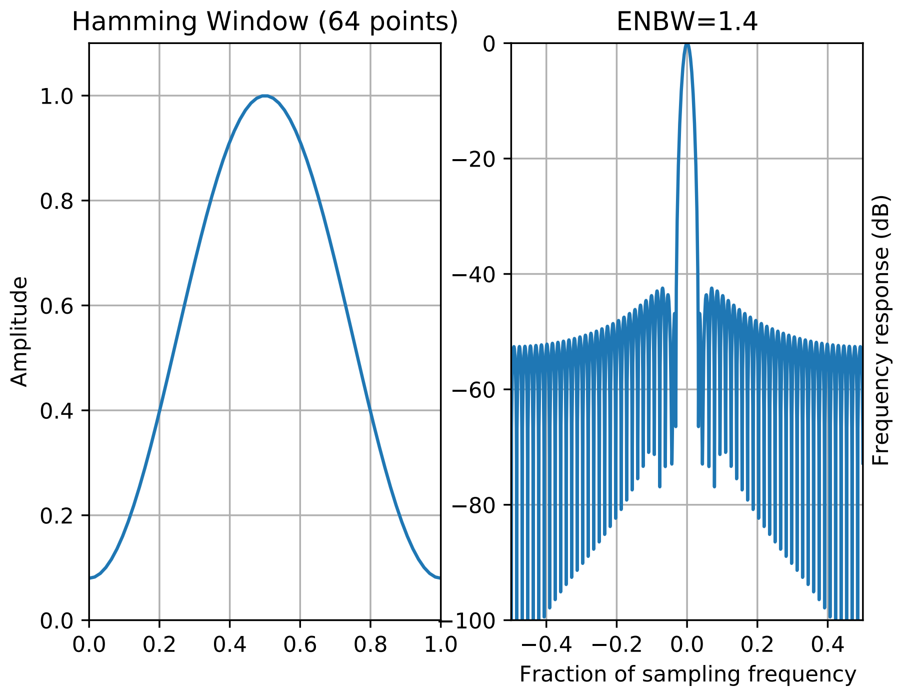
plot_window()
Plot the window in the time domain

```
from spectrum.window import Window
w = Window(64, name='hamming')
w.plot_window()
```

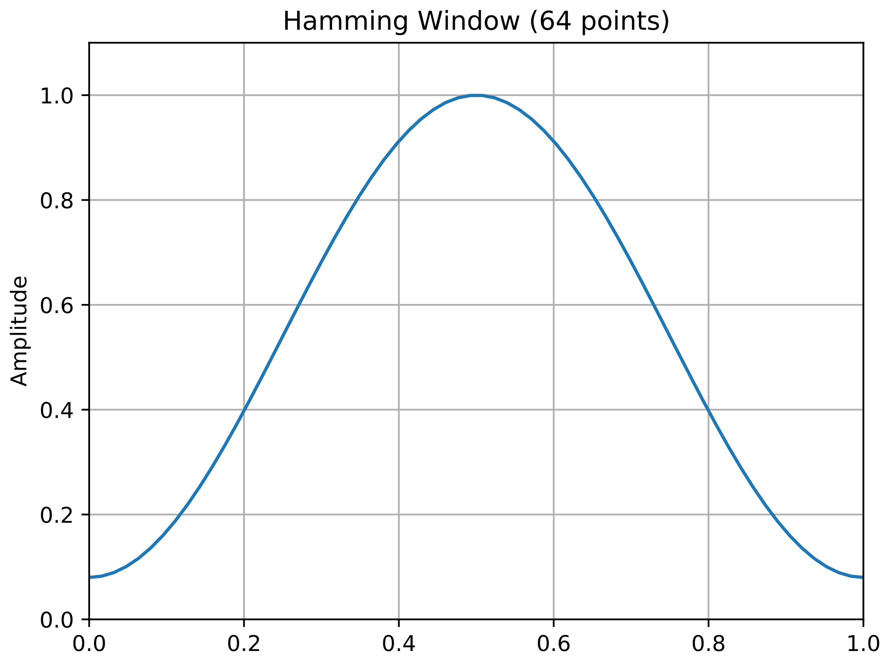
response
Getter for the frequency response. See *compute_response()*

create_window(*N*, *name=None*, ***kargs*)
Returns the N-point window given a valid name

Parameters



Hamming Window (64 points)
Amplitude
ENBW=1.4
Frequency response (dB)
Fraction of sampling frequency



Hamming Window (64 points)
Amplitude

- **N** (*int*) – window size
- **name** (*str*) – window name (default is *rectangular*). Valid names are stored in `window_names()`.
- **kargs** – optional arguments are:
  - *beta*: argument of the *window_kaiser()* function (default is 8.6)
  - *attenuation*: argument of the *window_chebwin()* function (default is 50dB)
  - ***alpha*: argument of the**
    1. *window_gaussian()* function (default is 2.5)
    2. *window_blackman()* function (default is 0.16)
    3. *window_poisson()* function (default is 2)
    4. *window_cauchy()* function (default is 3)
  - *mode*: argument *window_flattop()* function (default is *symmetric*, can be *periodic*)
  - *r*: argument of the *window_tukey()* function (default is 0.5).

The following windows have been simply wrapped from existing librairies like NumPy:

- **Rectangular**: *window_rectangle()*,
- **Bartlett** or Triangular: see *window_bartlett()*,
- **Hanning** or Hann: see *window_hann()*,
- **Hamming**: see *window_hamming()*,
- **Kaiser**: see *window_kaiser()*,
- **chebwin**: see *window_chebwin()*.

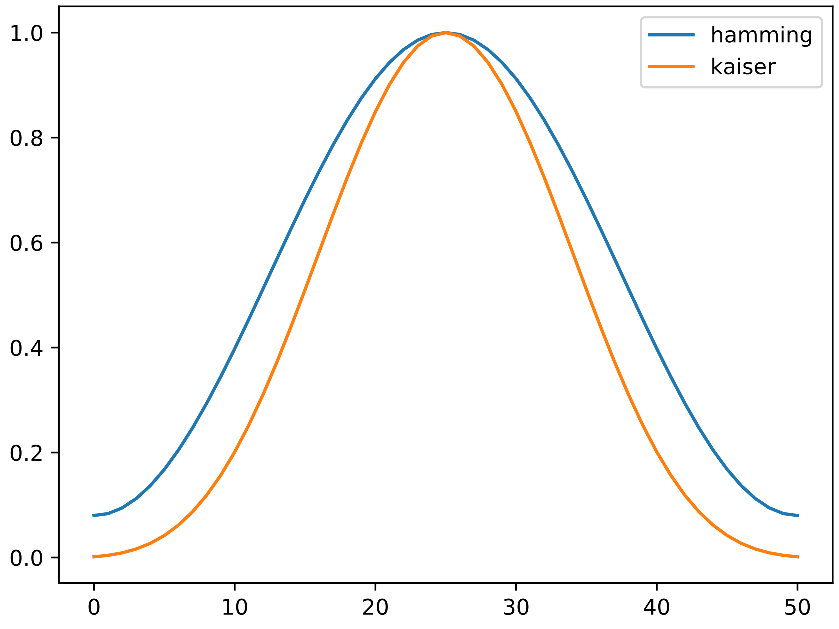
The following windows have been implemented from scratch:

- **Blackman**: See *window_blackman()*
- **Bartlett-Hann** : see *window_bartlett_hann()*
- **cosine or sine**: see *window_cosine()*
- **gaussian**: see *window_gaussian()*
- **Bohman**: see *window_bohman()*
- **Lanczos or sinc**: see *window_lanczos()*
- **Blackman Harris**: see *window_blackman_harris()*
- **Blackman Nuttall**: see *window_blackman_nuttall()*
- **Nuttall**: see *window_nuttall()*
- **Tukey**: see *window_tukey()*
- **Parzen**: see *window_parzen()*
- **Flattop**: see *window_flattop()*
- **Riesz**: see *window_riesz()*
- **Riemann**: see *window_riemann()*
- **Poisson**: see *window_poisson()*

- **Poisson-Hanning**: see *window_poisson_hanning()*

Todo: on request taylor, potter, Bessel, expo, rife-vincent, Kaiser-Bessel derived (KBD)

```
from pylab import plot, legend
from spectrum import create_window

data = create_window(51, 'hamming')
plot(data, label='hamming')
data = create_window(51, 'kaiser')
plot(data, label='kaiser')
legend()
```

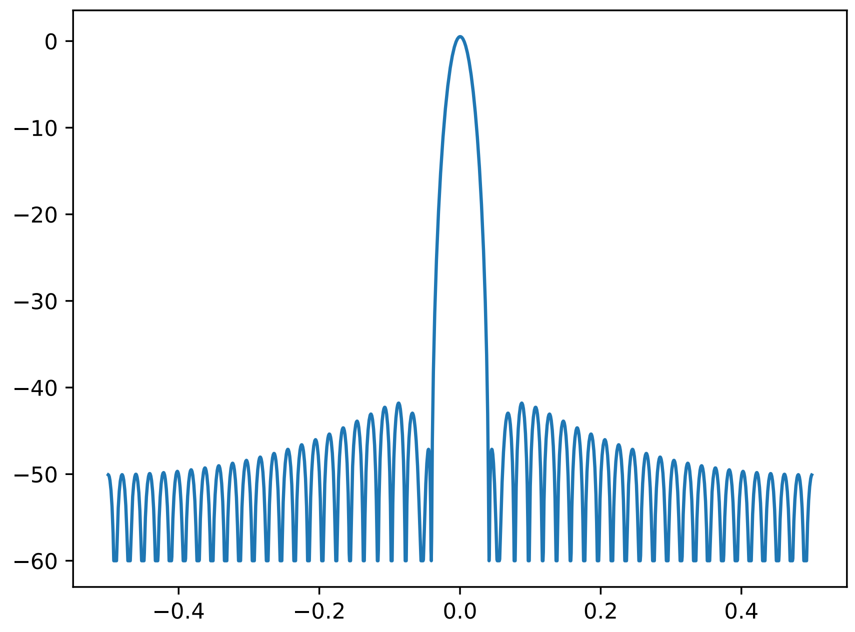


```
from pylab import plot, log10, linspace, fft, clip
from spectrum import create_window, fftshift

A = fft(create_window(51, 'hamming'), 2048) / 25.5
mag = abs(fftshift(A))
freq = linspace(-0.5,0.5,len(A))
response = 20 * log10(mag)
mindB = -60
response = clip(response,mindB,100)
plot(freq, response)
```

See also:

window_visu(), *Window()*, `spectrum.dpss`

enbw(*data*)



Computes the equivalent noise bandwidth

$$ENBW = N \frac{\sum_{n=1}^{N} w_n^2}{\left(\sum_{n=1}^{N} w_n\right)^2}$$

```
>>> from spectrum import create_window, enbw
>>> w = create_window(64, 'rectangular')
>>> enbw(w)
1.0
```

The following table contains the ENBW values for some of the implemented windows in this module (with N=16384). They have been double checked against litterature (Source: [Harris], [Marple]).

If not present, it means that it has not been checked.

name	ENBW	litterature
rectangular	1.	1.
triangle	1.3334	1.33
Hann	1.5001	1.5
Hamming	1.3629	1.36
blackman	1.7268	1.73
kaiser	1.7	
blackmanharris,4	2.004	2.
riesz	1.2000	1.2
riemann	1.32	1.3
parzen	1.917	1.92
tukey 0.25	1.102	1.1
bohman	1.7858	1.79
poisson 2	1.3130	1.3
hanningpoisson 0.5	1.609	1.61
cauchy	1.489	1.48
lanczos	1.3	

window_bartlett (*N*)

Bartlett window (wrapping of numpy.bartlett) also known as Fejer

Parameters N (*int*) – window length

The Bartlett window is defined as

$$w(n) = \frac{2}{N-1}\left(\frac{N-1}{2} - \left|n - \frac{N-1}{2}\right|\right)$$

```
from spectrum import window_visu
window_visu(64, 'bartlett')
```

See also:

numpy.bartlett, *create_window()*, *Window*.

window_bartlett_hann (*N*)

Bartlett-Hann window

Parameters N – window length

$$w(n) = a_0 + a_1 \left| \frac{n}{N-1} - \frac{1}{2} \right| - a_2 \cos \left(\frac{2\pi n}{N-1} \right)$$

with $a_0 = 0.62$, $a_1 = 0.48$ and $a_2 = 0.38$

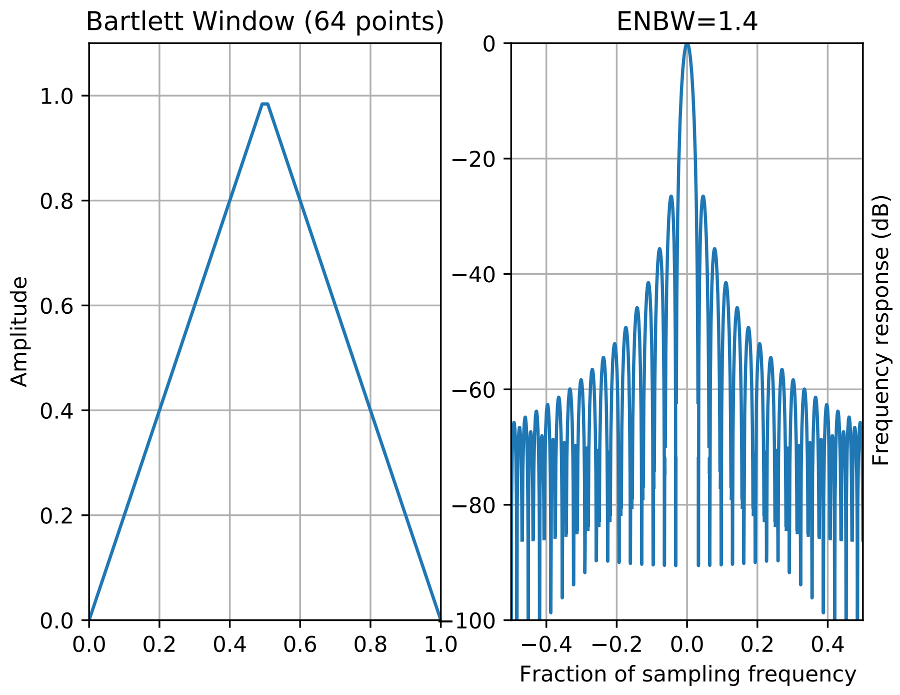
```
from spectrum import window_visu
window_visu(64, 'bartlett_hann')
```

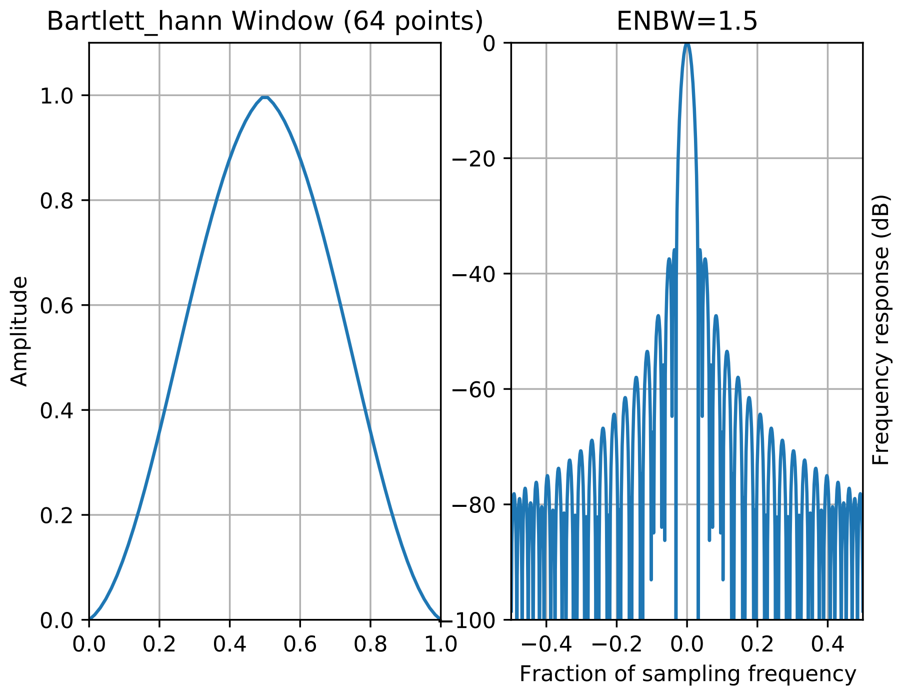
See also:

create_window(), *Window*

window_blackman (*N*, *alpha=0.16*)

Blackman window


Bartlett Window (64 points)
ENBW=1.4
Amplitude
Frequency response (dB)
Fraction of sampling frequency


Bartlett_hann Window (64 points)
ENBW=1.5
Amplitude
Frequency response (dB)
Fraction of sampling frequency

Parameters N – window length

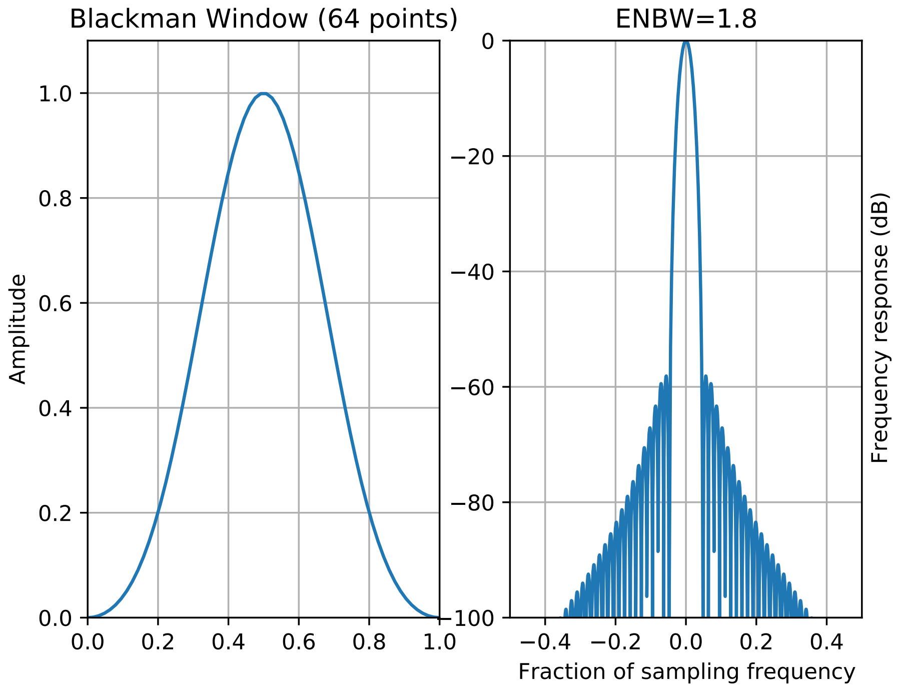
$$a_0 - a_1 \cos(\frac{2\pi n}{N-1}) + a_2 \cos(\frac{4\pi n}{N-1})$$

with

$$a_0 = (1-\alpha)/2, a_1 = 0.5, a_2 = \alpha/2 \text{ and } \alpha = 0.16$$

When $\alpha = 0.16$, this is the unqualified Blackman window with $a_0 = 0.48$ and $a_2 = 0.08$.

```
from spectrum import window_visu
window_visu(64, 'blackman')
```



Note: Although Numpy implements a blackman window for $\alpha = 0.16$, this implementation is valid for any α.

See also:

numpy.blackman, *create_window()*, *Window*

window_blackman_harris(*N*)

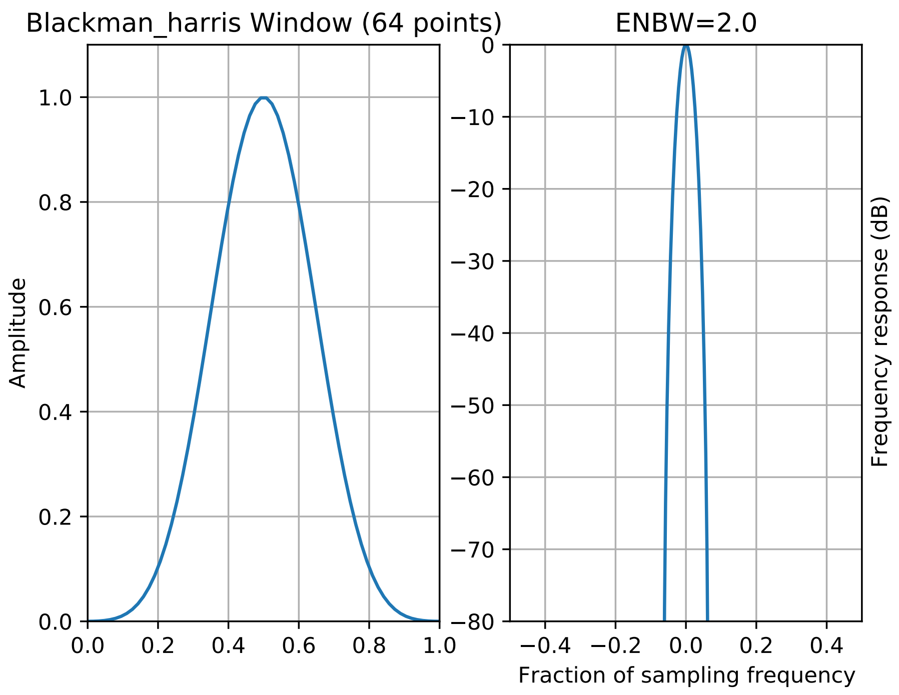
Blackman Harris window

Parameters N – window length

$$w(n) = a_0 - a_1 \cos\left(\frac{2\pi n}{N-1}\right) + a_2 \cos\left(\frac{4\pi n}{N-1}\right) - a_3 \cos\left(\frac{6\pi n}{N-1}\right)$$

coeff	value
a_0	0.35875
a_1	0.48829
a_2	0.14128
a_3	0.01168

```
from spectrum import window_visu
window_visu(64, 'blackman_harris', mindB=-80)
```



See also:

spectrum.window.create_window()

See also:

create_window(), *Window*

window_blackman_nuttall(*N*)

Blackman Nuttall window

returns a minimum, 4-term Blackman-Harris window. The window is minimum in the sense that its maximum sidelobes are minimized. The coefficients for this window differ from the Blackman-Harris window coefficients and produce slightly lower sidelobes.

Parameters N – window length

$$w(n) = a_0 - a_1 \cos\left(\frac{2\pi n}{N-1}\right) + a_2 \cos\left(\frac{4\pi n}{N-1}\right) - a_3 \cos\left(\frac{6\pi n}{N-1}\right)$$

with $a_0 = 0.3635819$, $a_1 = 0.4891775$, $a_2 = 0.1365995$ and $0_3 = .0106411$

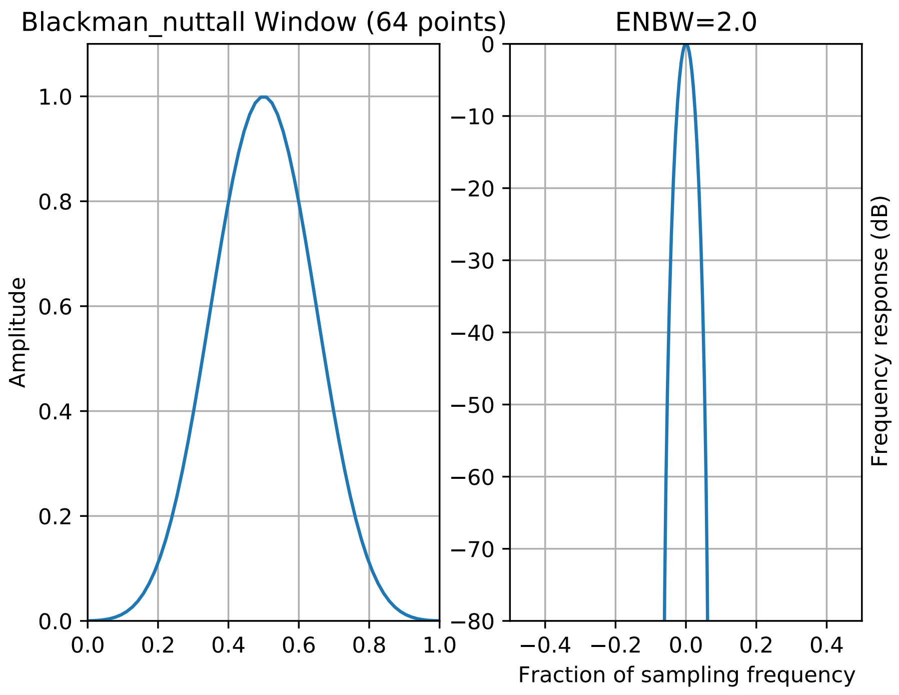
```
from spectrum import window_visu
window_visu(64, 'blackman_nuttall', mindB=-80)
```

See also:

spectrum.window.create_window()

See also:

create_window(), *Window*



window_bohman (*N*)

Bohman tapering window

Parameters **N** – window length

$$w(n) = (1 - |x|)\cos(\pi |x|) + \frac{1}{\pi}\sin(\pi |x|)$$

where x is a length N vector of linearly spaced values between -1 and 1.

```
from spectrum import window_visu
window_visu(64, 'bohman')
```

See also:

create_window(), *Window*

window_cauchy (*N*, *alpha=3*)

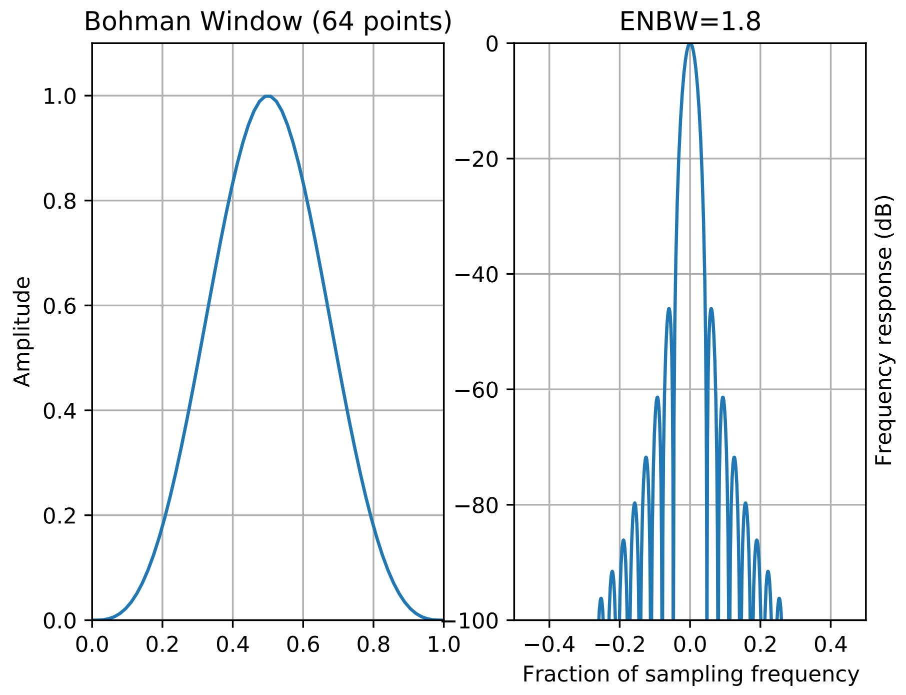
Cauchy tapering window

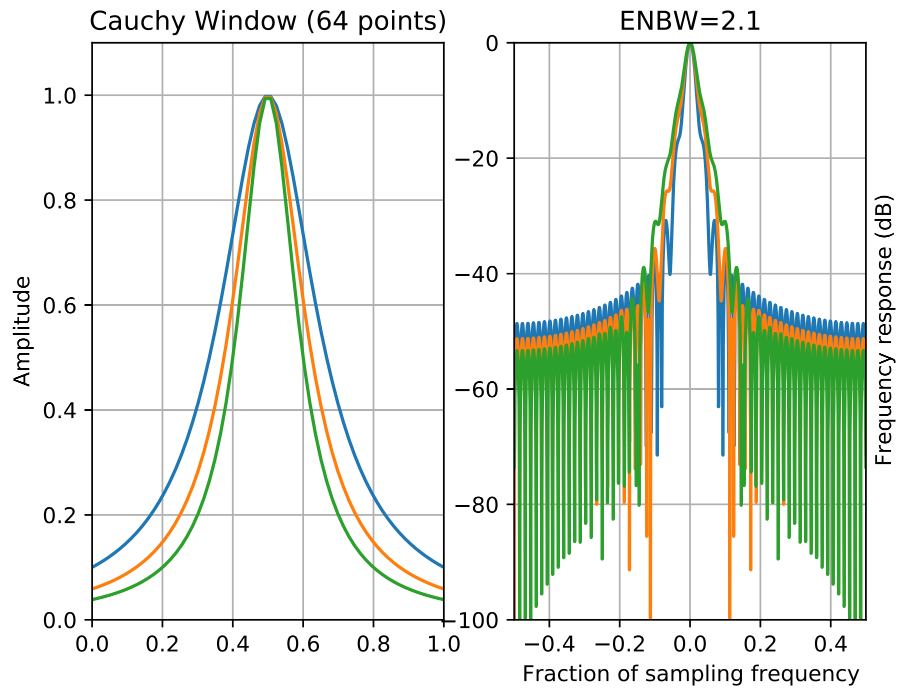
Parameters

- **N** (*int*) – window length
- **alpha** (*float*) – parameter of the poisson window

$$w(n) = \frac{1}{1 + \left(\frac{\alpha * n}{N/2}\right) * *2}$$

```
from spectrum import window_visu
window_visu(64, 'cauchy', alpha=3)
window_visu(64, 'cauchy', alpha=4)
window_visu(64, 'cauchy', alpha=5)
```


Bohman Window (64 points)
ENBW=1.8
Amplitude
Frequency response (dB)
Fraction of sampling frequency
1.0
0.8
0.6
0.4
0.2
0.0
0
−20
−40
−60
−80
−100
−0.4
−0.2
0.0
0.2
0.4


Cauchy Window (64 points)
ENBW=2.1
Amplitude
Frequency response (dB)
Fraction of sampling frequency
1.0
0.8
0.6
0.4
0.2
0.0
0
−20
−40
−60
−80
−100
−0.4
−0.2
0.0
0.2
0.4

See also:

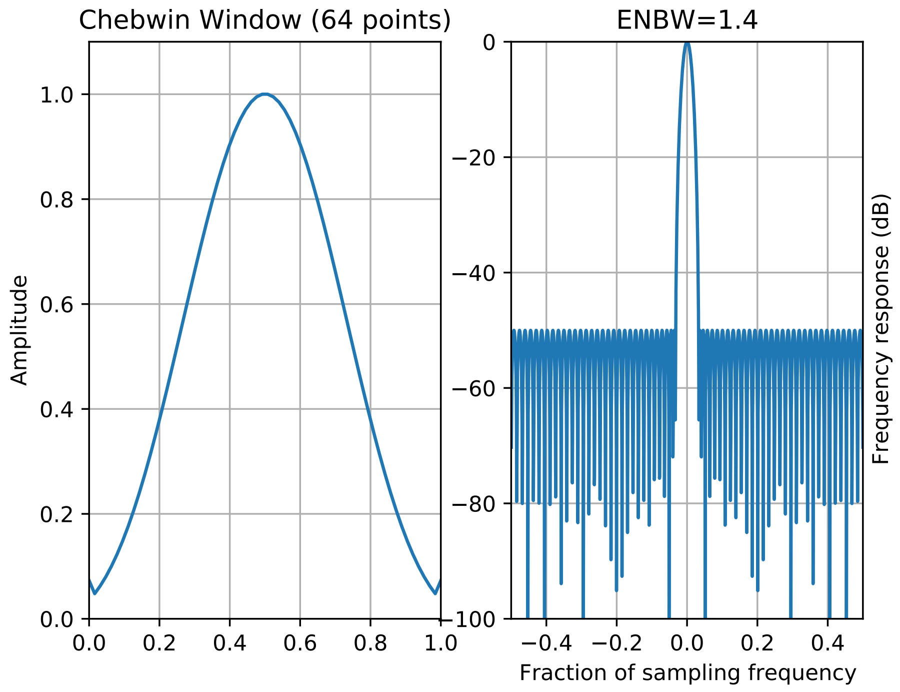
window_poisson(), *window_hann()*

window_chebwin (*N*, *attenuation=50*)

Cheb window

Parameters N – window length

```
from spectrum import window_visu
window_visu(64, 'chebwin', attenuation=50)
```



See also:

scipy.signal.chebwin, *create_window()*, *Window*

window_cosine (*N*)

Cosine tapering window also known as sine window.

Parameters N – window length

$$w(n) = \cos\left(\frac{\pi n}{N-1} - \frac{\pi}{2}\right) = \sin\left(\frac{\pi n}{N-1}\right)$$

```
from spectrum import window_visu
window_visu(64, 'cosine')
```

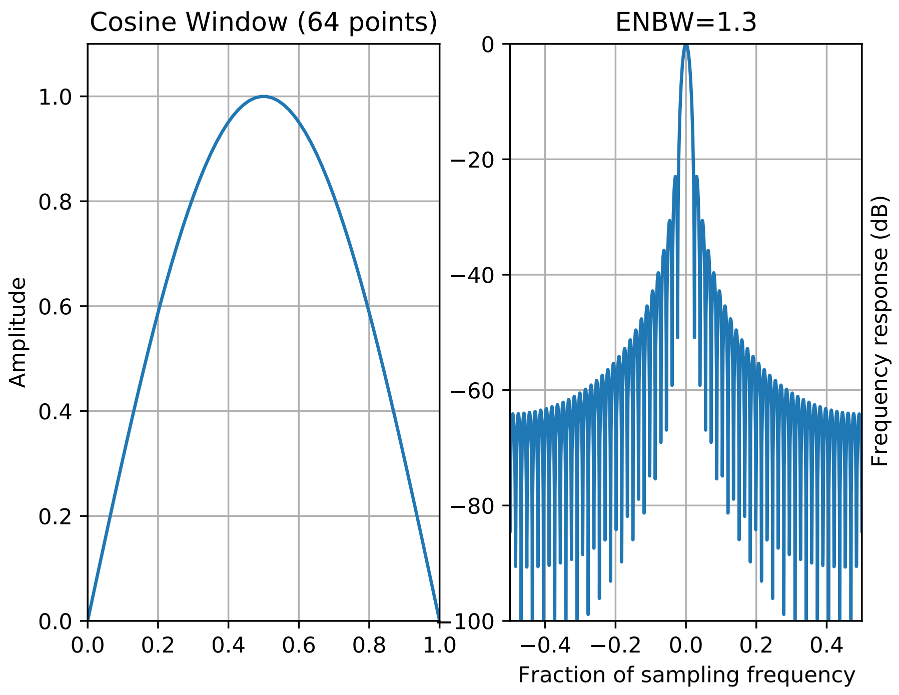
See also:

create_window(), *Window*

window_flattop (*N*, *mode='symmetric'*, *precision=None*)

Flat-top tapering window

Returns symmetric or periodic flat top window.



Parameters

- **N** – window length
- **mode** – way the data are normalised. If mode is *symmetric*, then divide n by N-1. IF mode is *periodic*, divide by N, to be consistent with octave code.

When using windows for filter design, the *symmetric* mode should be used (default). When using windows for spectral analysis, the *periodic* mode should be used. The mathematical form of the flat-top window in the symmetric case is:

$$w(n) = a_0 - a_1 \cos\left(\frac{2\pi n}{N-1}\right) + a_2 \cos\left(\frac{4\pi n}{N-1}\right) - a_3 \cos\left(\frac{6\pi n}{N-1}\right) + a_4 \cos\left(\frac{8\pi n}{N-1}\right)$$

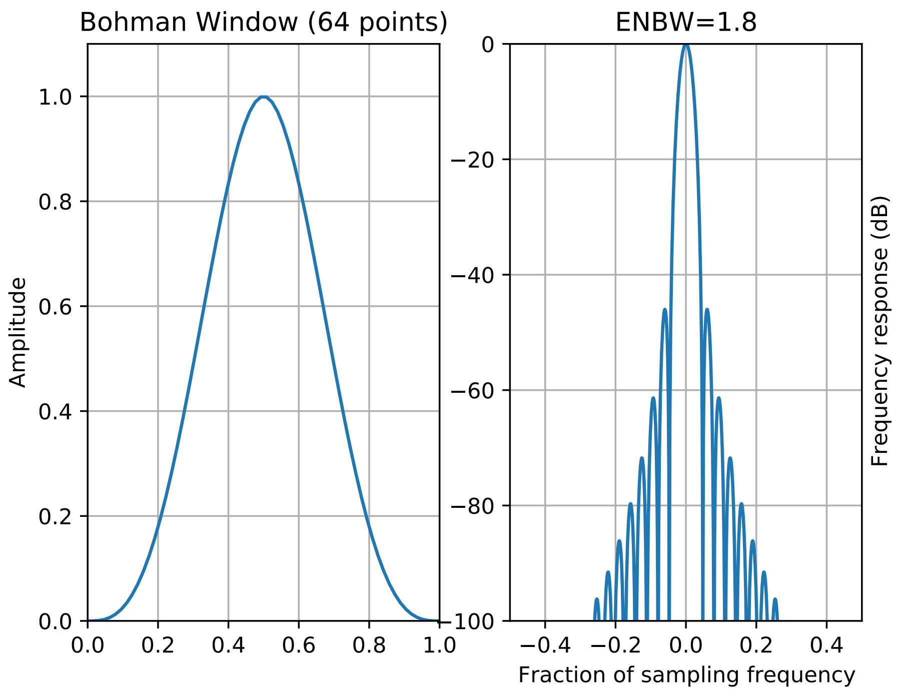
coeff	value
a0	0.21557895
a1	0.41663158
a2	0.277263158
a3	0.083578947
a4	0.006947368

```
from spectrum import window_visu
window_visu(64, 'bohman')
```

See also:

create_window(), *Window*

window_gaussian (*N*, *alpha=2.5*)
Gaussian window



Parameters **N** – window length

$$\exp^{-0.5\left(\sigma\frac{n}{N/2}\right)^2}$$

with $\frac{N-1}{2} \leq n \leq \frac{N-1}{2}$.

Note: N-1 is used to be in agreement with octave convention. The ENBW of 1.4 is also in agreement with [Harris]

```
from spectrum import window_visu
window_visu(64, 'gaussian', alpha=2.5)
```

See also:

scipy.signal.gaussian, *create_window()*

window_hamming(*N*)

Hamming window

Parameters **N** – window length

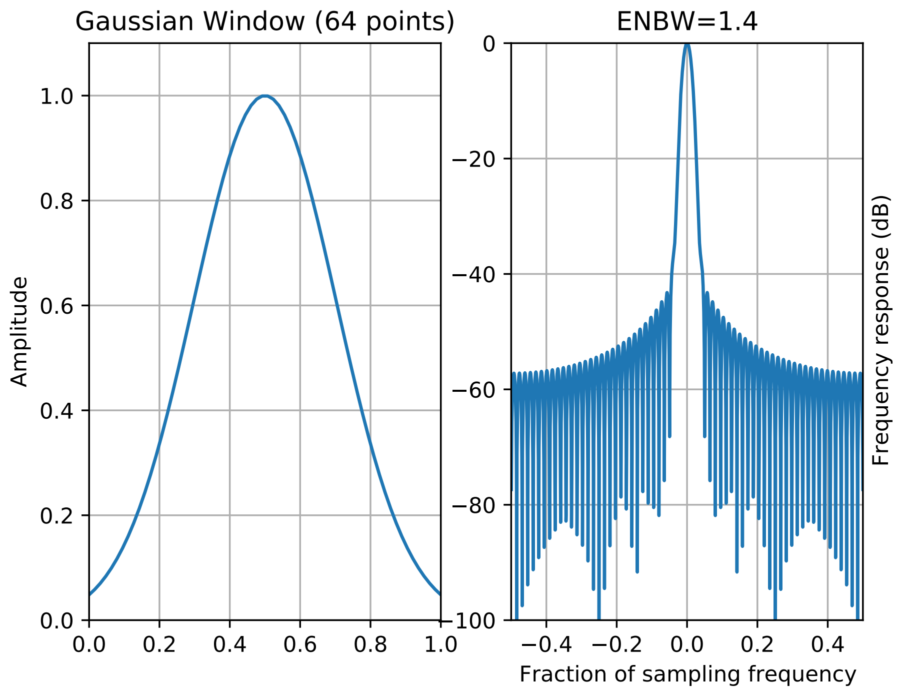
The Hamming window is defined as

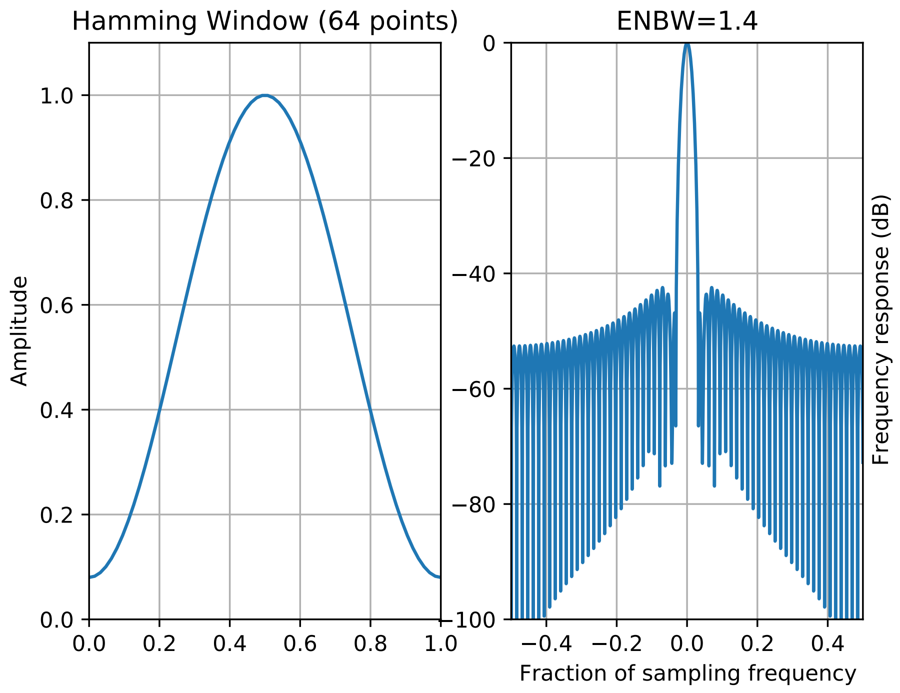
$$0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right) \qquad 0 \leq n \leq M-1$$

```
from spectrum import window_visu
window_visu(64, 'hamming')
```

See also:

numpy.hamming, *create_window()*, *Window*.


Gaussian Window (64 points)
ENBW=1.4
Amplitude
Frequency response (dB)
Fraction of sampling frequency


Hamming Window (64 points)
ENBW=1.4
Amplitude
Frequency response (dB)
Fraction of sampling frequency

window_hann (*N*)

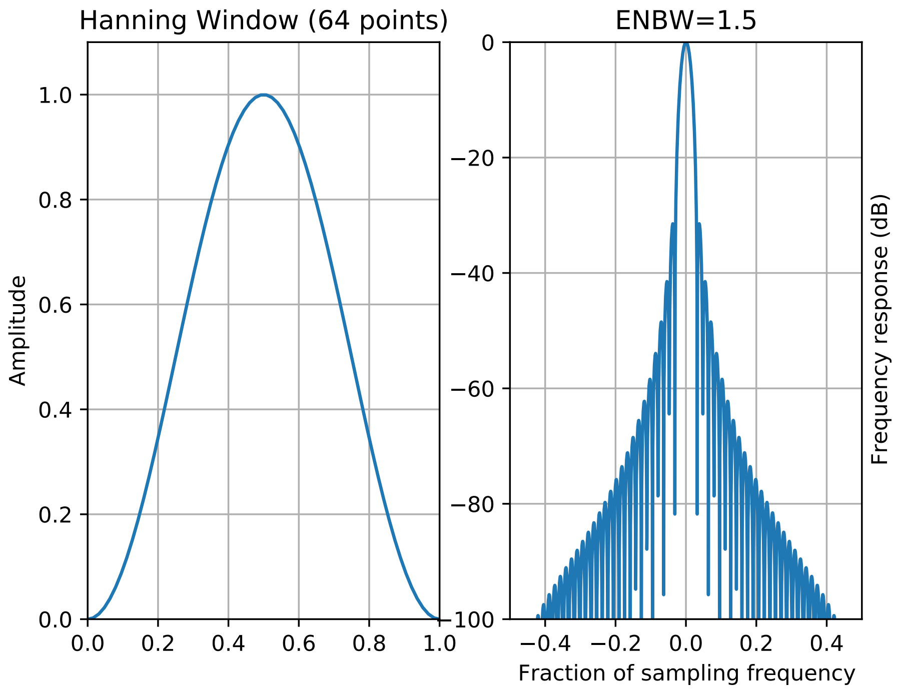
Hann window (or Hanning). (wrapping of numpy.bartlett)

Parameters **N** (*int*) – window length

The Hanning window is also known as the Cosine Bell. Usually, it is called Hann window, to avoid confusion with the Hamming window.

$$w(n) = 0.5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right) \qquad 0 \leq n \leq M-1$$

```
from spectrum import window_visu
window_visu(64, 'hanning')
```



See also:

numpy.hanning, *create_window()*, *Window*.

window_kaiser (*N*, *beta=8.6*, *method='numpy'*)

Kaiser window

Parameters

- **N** – window length
- **beta** – kaiser parameter (default is 8.6)

To obtain a Kaiser window that designs an FIR filter with sidelobe attenuation of α dB, use the following β where $\beta = \pi\alpha$.

$$w_n = \frac{I_0\left(\pi\alpha\sqrt{1 - \left(\frac{2n}{M} - 1\right)^2}\right)}{I_0(\pi\alpha)}$$

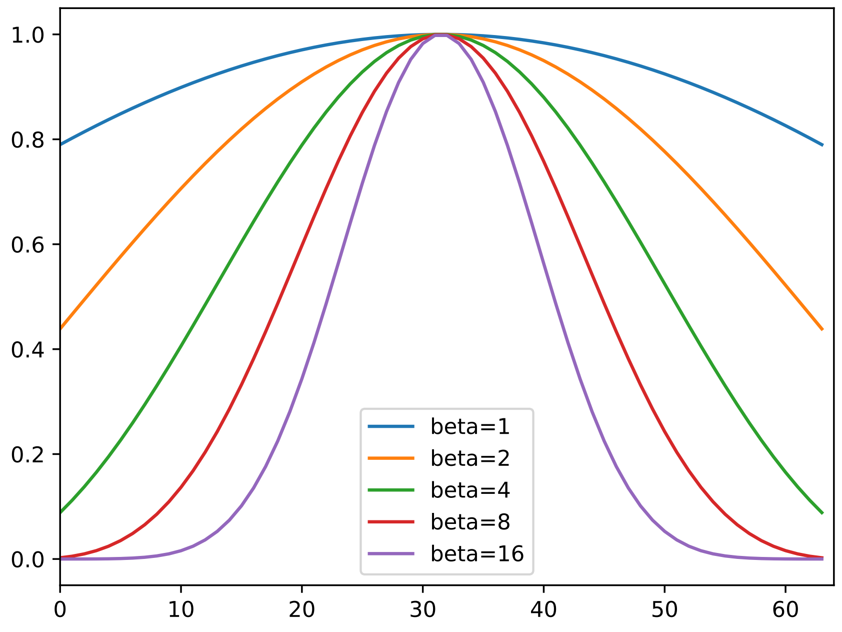
where

- I_0 is the zeroth order Modified Bessel function of the first kind.
- α is a real number that determines the shape of the window. It determines the trade-off between main-lobe width and side lobe level.
- the length of the sequence is N=M+1.

The Kaiser window can approximate many other windows by varying the β parameter:

beta	Window shape
0	Rectangular
5	Similar to a Hamming
6	Similar to a Hanning
8.6	Similar to a Blackman

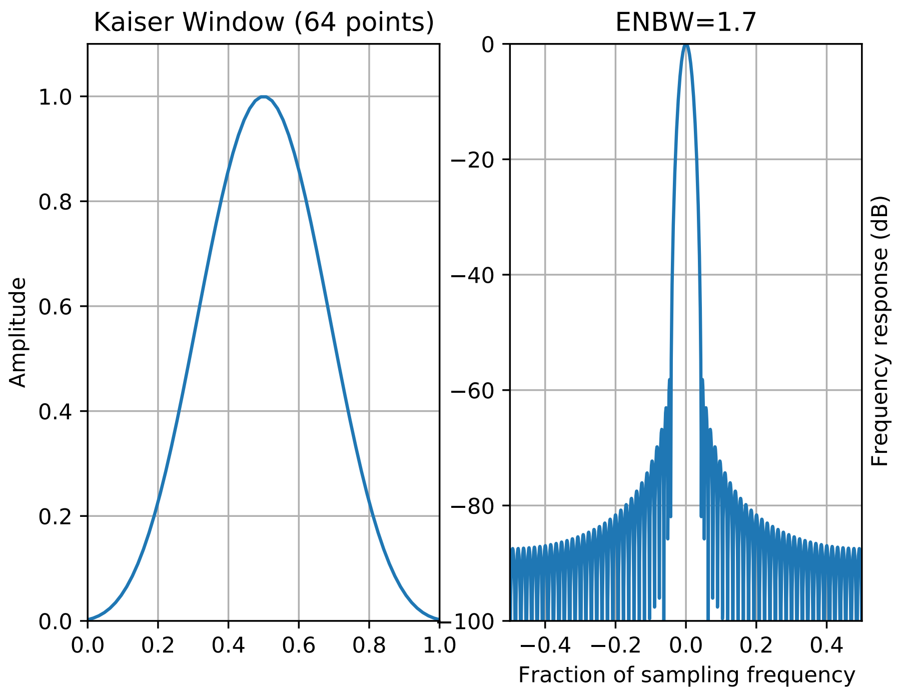
```
from pylab import plot, legend, xlim
from spectrum import window_kaiser
N = 64
for beta in [1,2,4,8,16]:
    plot(window_kaiser(N, beta), label='beta='+str(beta))
xlim(0,N)
legend()
```



```
from spectrum import window_visu
window_visu(64, 'kaiser', beta=8.)
```

See also:

numpy.kaiser, *spectrum.window.create_window()*



window_lanczos(N)

Lanczos window also known as sinc window.

Parameters **N** – window length

$$w(n) = sinc \left(\frac{2n}{N-1} - 1 \right)$$

```
from spectrum import window_visu
window_visu(64, 'lanczos')
```

See also:

create_window(), *Window*

window_nuttall(N)

Nuttall tapering window

Parameters **N** – window length

$$w(n) = a_0 - a_1 \cos\left(\frac{2\pi n}{N-1}\right) + a_2 \cos\left(\frac{4\pi n}{N-1}\right) - a_3 \cos\left(\frac{6\pi n}{N-1}\right)$$

with $a_0 = 0.355768$, $a_1 = 0.487396$, $a_2 = 0.144232$ and $a_3 = 0.012604$

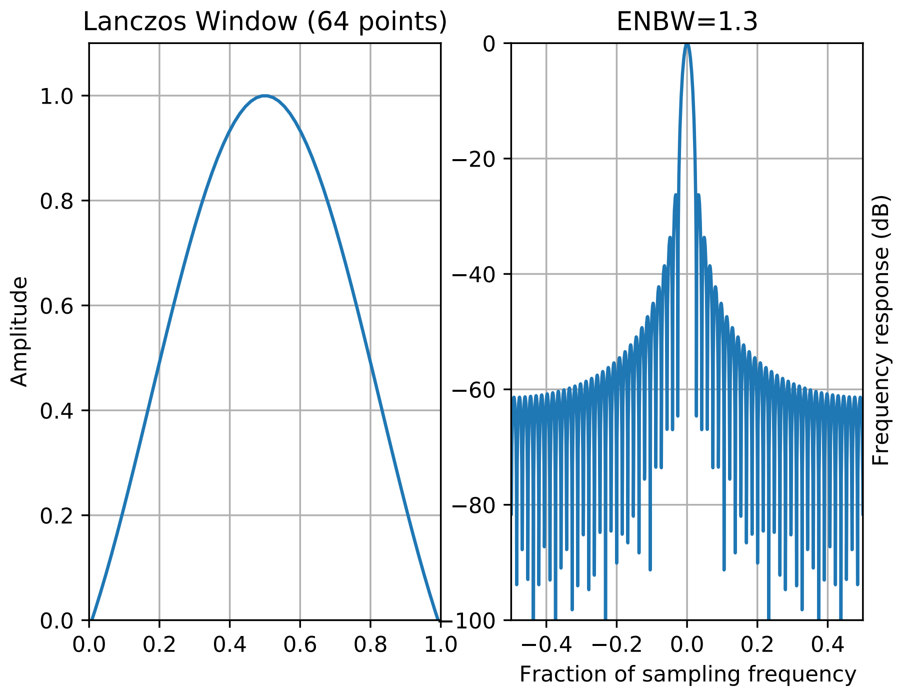
```
from spectrum import window_visu
window_visu(64, 'nuttall', mindB=-80)
```

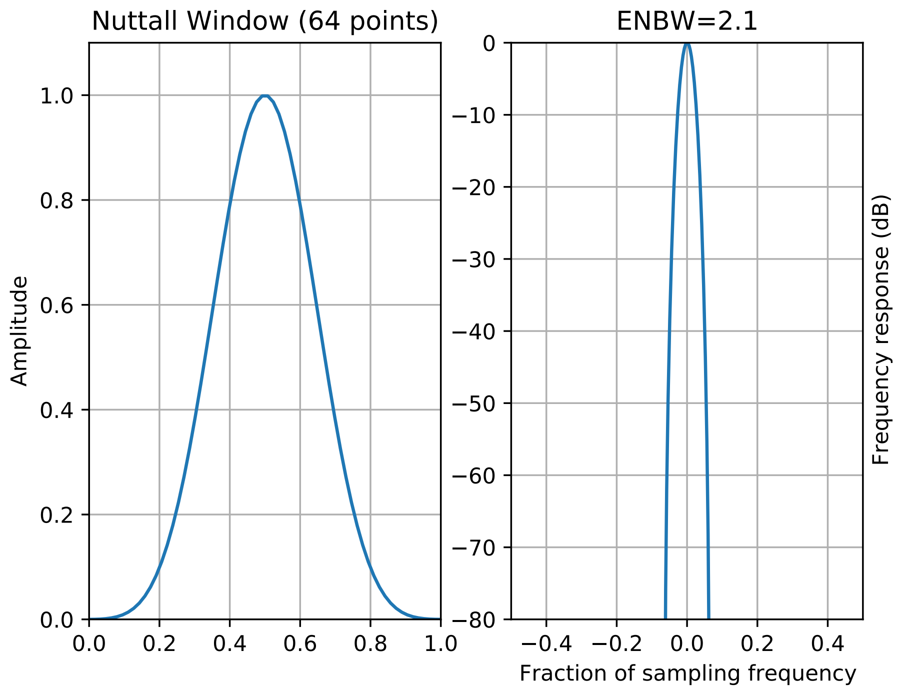
See also:

create_window(), *Window*

window_parzen(N)

Parsen tapering window (also known as de la Valle-Poussin)


Lanczos Window (64 points)
ENBW=1.3
Amplitude
Frequency response (dB)
Fraction of sampling frequency
0.0
0.2
0.4
0.6
0.8
1.0
0
−20
−40
−60
−80
−100
−0.4
−0.2
0.0
0.2
0.4


Nuttall Window (64 points)
ENBW=2.1
Amplitude
Frequency response (dB)
Fraction of sampling frequency
0.0
0.2
0.4
0.6
0.8
1.0
0
−10
−20
−30
−40
−50
−60
−70
−80
−0.4
−0.2
0.0
0.2
0.4

Parameters **N** – window length

Parzen windows are piecewise cubic approximations of Gaussian windows. Parzen window sidelobes fall off as $1/\omega^4$.

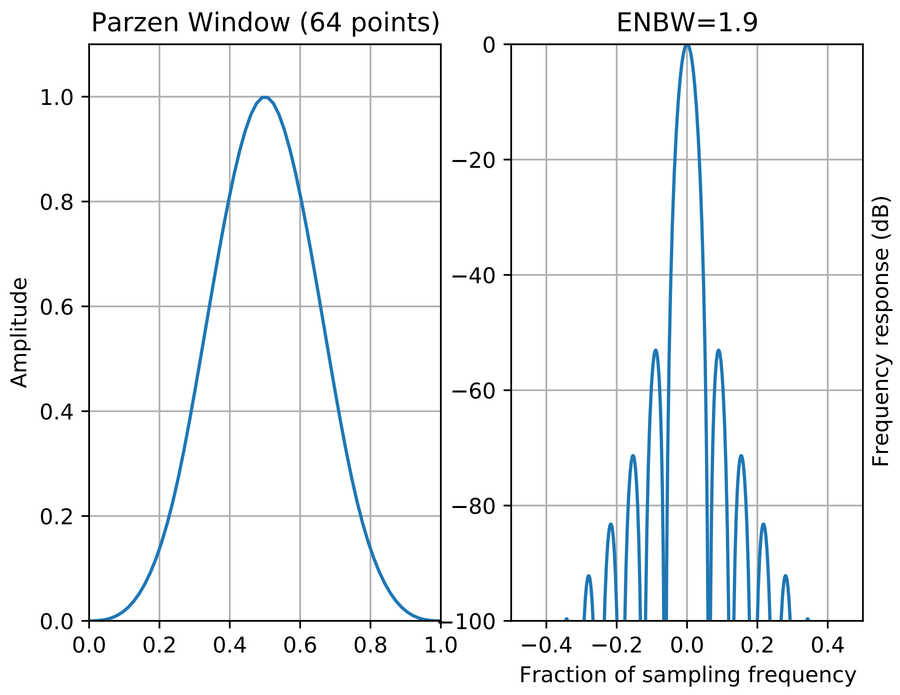
if $0 \leq |x| \leq (N-1)/4$:

$$w(n) = 1 - 6\left(\frac{|n|}{N/2}\right)^2 + 6\left(\frac{|n|}{N/2}\right)^3$$

if $(N-1)/4 \leq |x| \leq (N-1)/2$

$$w(n) = 2\left(1 - \frac{|n|}{N/2}\right)^3$$

```
from spectrum import window_visu
window_visu(64, 'parzen')
```



See also:

create_window(), *Window*

window_poisson (*N*, *alpha=2*)

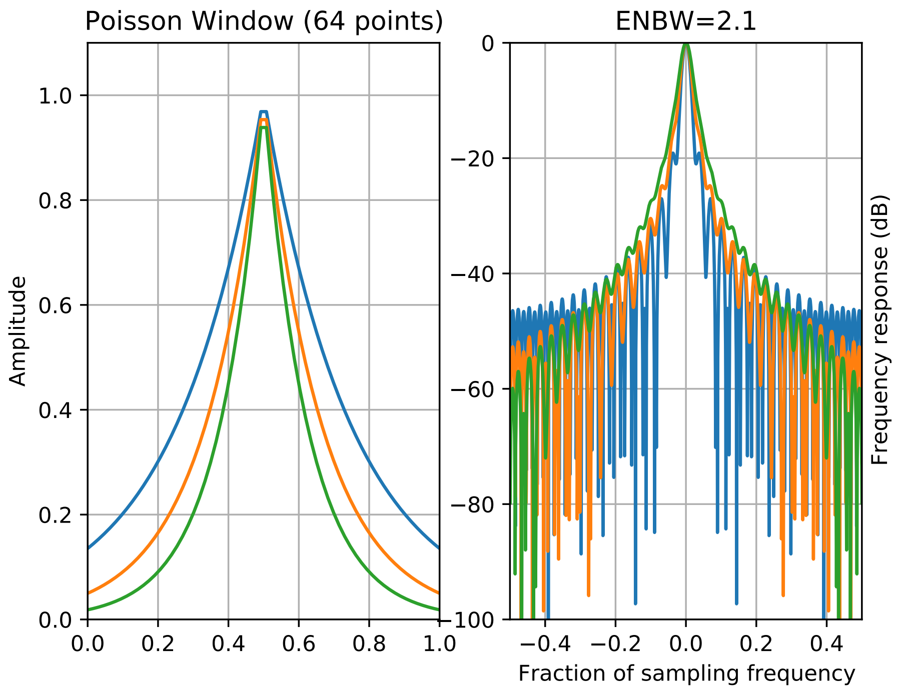
Poisson tapering window

Parameters **N** (*int*) – window length

$$w(n) = \exp^{-\alpha\frac{|n|}{N/2}}$$

with $-N/2 \leq n \leq N/2$.

```
from spectrum import window_visu
window_visu(64, 'poisson')
window_visu(64, 'poisson', alpha=3)
window_visu(64, 'poisson', alpha=4)
```



See also:

create_window(), *Window*

window_poisson_hanning(*N*, *alpha=2*)

Hann-Poisson tapering window

This window is constructed as the product of the Hanning and Poisson windows. The parameter **alpha** is the Poisson parameter.

Parameters

- **N** (*int*) – window length
- **alpha** (*float*) – parameter of the poisson window

```
from spectrum import window_visu
window_visu(64, 'poisson_hanning', alpha=0.5)
window_visu(64, 'poisson_hanning', alpha=1)
window_visu(64, 'poisson_hanning')
```

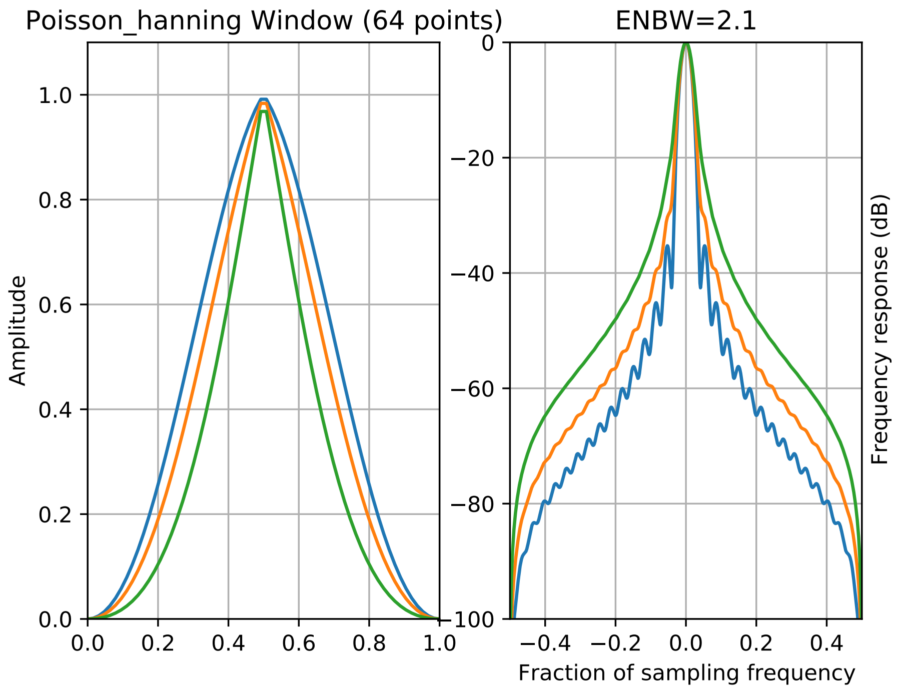
See also:

window_poisson(), *window_hann()*

window_rectangle(*N*)

Kaiser window

Parameters **N** – window length



```
from spectrum import window_visu
window_visu(64, 'rectangle')
```

window_riemann (N)

Riemann tapering window

Parameters **N** (*int*) – window length

$$w(n) = 1 - \left| \frac{n}{N/2} \right|^2$$

with $-N/2 \leq n \leq N/2$.

```
from spectrum import window_visu
window_visu(64, 'riesz')
```

See also:

create_window(), *Window*

window_riesz (N)

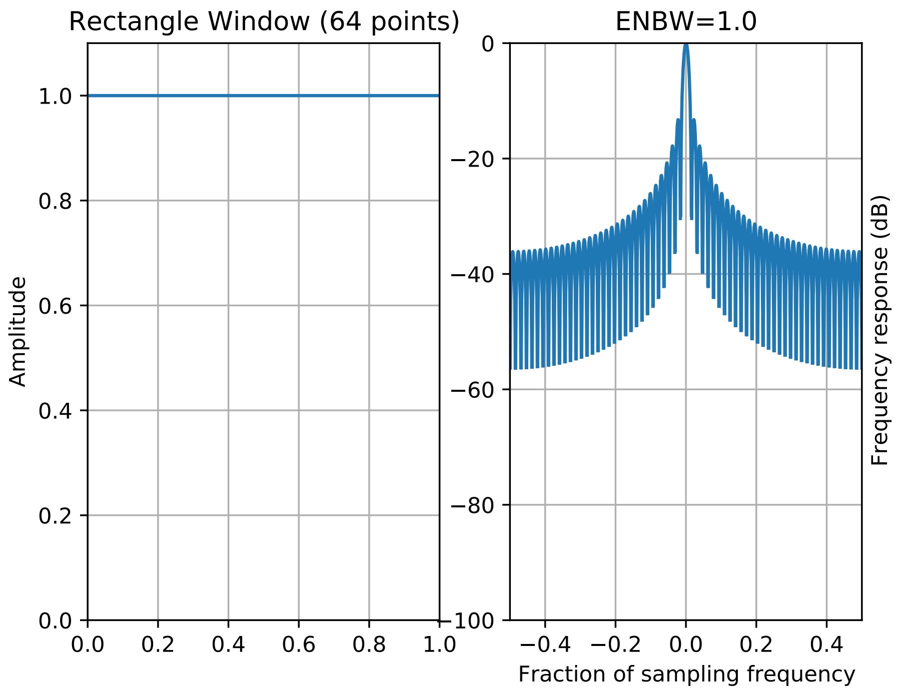
Riesz tapering window

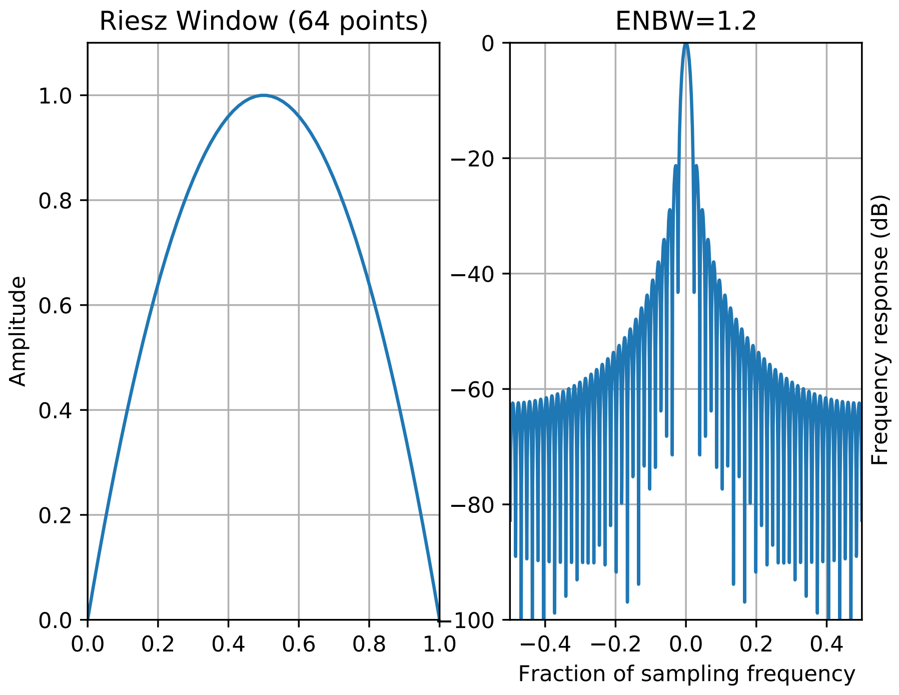
Parameters **N** – window length

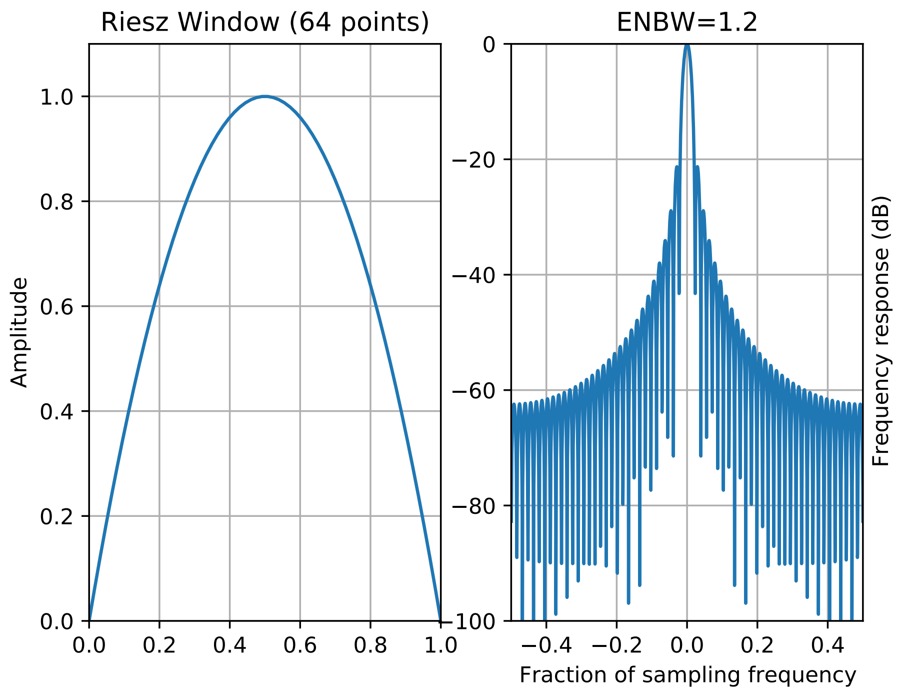
$$w(n) = 1 - \left| \frac{n}{N/2} \right|^2$$

with $-N/2 \leq n \leq N/2$.

```
from spectrum import window_visu
window_visu(64, 'riesz')
```


Rectangle Window (64 points)
ENBW=1.0
Amplitude
Frequency response (dB)
Fraction of sampling frequency


Riesz Window (64 points)
ENBW=1.2
Amplitude
Frequency response (dB)
Fraction of sampling frequency



See also:

create_window(), *Window*

window_taylor (*N*, *nbar=4*, *sll=-30*)

Taylor tapering window

Taylor windows allows you to make tradeoffs between the mainlobe width and sidelobe level (sll).

Implemented as described by Carrara, Goodman, and Majewski in 'Spotlight Synthetic Aperture Radar: Signal Processing Algorithms' Pages 512-513

Parameters

- **N** – window length
- **nbar** (*float*) –
- **sll** (*float*) –

The default values gives equal height sidelobes (nbar) and maximum sidelobe level (sll).

Warning: not implemented

See also:

create_window(), *Window*

window_tukey (*N*, *r=0.5*)

Tukey tapering window (or cosine-tapered window)

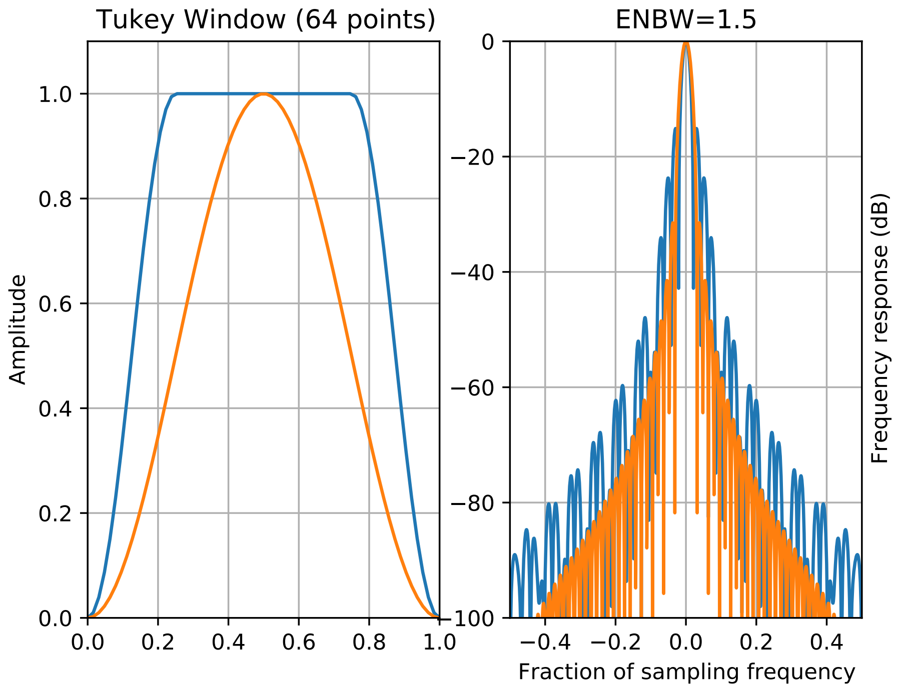
Parameters

- **N** – window length

- **r** – defines the ratio between the constant section and the cosine section. It has to be between 0 and 1.

The function returns a Hanning window for *r=0* and a full box for *r=1*.

```
from spectrum import window_visu
window_visu(64, 'tukey')
window_visu(64, 'tukey', r=1)
```



$$0.5(1 + cos(2pi/r(x - r/2)))for0 <= x < r/2$$

$$0.5(1 + cos(2pi/r(x - 1 + r/2)))forx >= 1 - r/2$$

See also:

create_window(), *Window*

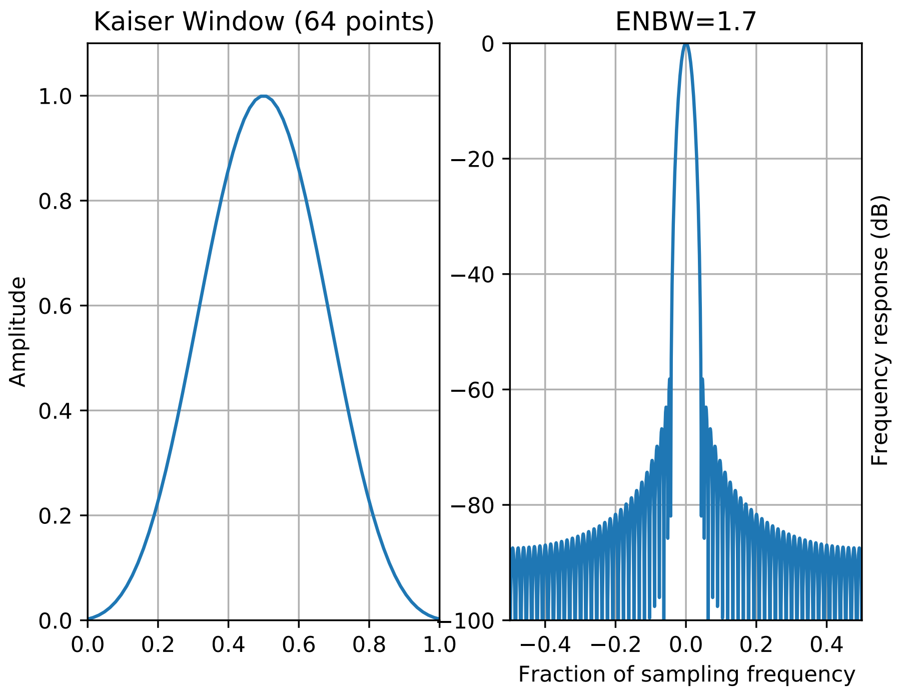
window_visu (*N=51*, *name='hamming'*, ***kargs*)
A Window visualisation tool

Parameters

- **N** – length of the window
- **name** – name of the window
- **NFFT** – padding used by the FFT
- **mindB** – the minimum frequency power in dB
- **maxdB** – the maximum frequency power in dB
- **kargs** – optional arguments passed to *create_window()*

This function plot the window shape and its equivalent in the Fourier domain.

```
from spectrum import window_visu
window_visu(64, 'kaiser', beta=8.)
```



5.2 Multitapering

Multitapering method.

This module provides a multi-tapering method for spectral estimation. The computation of the tapering windows being computationally expensive, a C module is used to compute the tapering window (see `spectrum.mtm.dpss()`).

The estimation itself can be performed with the `spectrum.mtm.pmtm()` function

class MultiTapering(*data*, *NW=None*, *k=None*, *NFFT=None*, *e=None*, *v=None*, *method='adapt'*, *scale_by_freq=True*, *sampling=1*)

See `pmtm()` for details

dpss(*N*, *NW=None*, *k=None*)

Discrete prolate spheroidal (Slepian) sequences

Calculation of the Discrete Prolate Spheroidal Sequences also known as the slepian sequences, and the corresponding eigenvalues.

Parameters

- **N** (*int*) – desired window length
- **NW** (*float*) – The time half bandwidth parameter (typical values are 2.5,3,3.5,4).
- **k** (*int*) – returns the first k Slepian sequences. If *k* is not provided, *k* is set to *NW*2*.

Returns

- tapers, a matrix of tapering windows. Matrix is a N by *k* (k is the number of windows)
- eigen, a vector of eigenvalues of length *k*

The discrete prolate spheroidal or Slepian sequences derive from the following time-frequency concentration problem. For all finite-energy sequences index limited to some set , which sequence maximizes the following ratio:

$$\lambda = \frac{\int_{-W}^{W} |X(f)|^2 df}{\int_{-F_s/2}^{F_s/2} |X(f)|^2 df}$$

where F_s is the sampling frequency and $|W| < F_s/2$. This ratio determines which index-limited sequence has the largest proportion of its energy in the band $[-W, W]$ with $0 < \lambda < 1$. The sequence maximizing the ratio is the first discrete prolate spheroidal or Slepian sequence. The second Slepian sequence maximizes the ratio and is orthogonal to the first Slepian sequence. The third Slepian sequence maximizes the ratio of integrals and is orthogonal to both the first and second Slepian sequences and so on.

Note: Note about the implementation. Since the slepian generation is computationally expensive, we use a C implementation based on the C code written by Lees as published in:

> Lees, J. M. and J. Park (1995): Multiple-taper spectral analysis: A stand-alone C-subroutine: Computers & Geology: 21, 199-236.

However, the original C code has been trimmed. Indeed, we only require the multitap function (that depends on jtridib, jtinvit functions only).

```
from spectrum import *
from pylab import *
N = 512
[w, eigens] = dpss(N, 2.5, 4)
plot(w)
title('Slepian Sequences N=%s, NW=2.5' % N)
axis([0, N, -0.15, 0.15])
legend(['1st window','2nd window','3rd window','4th window'])
```

Windows are normalised:

$$\sum_k h_k h_k = 1$$

References [Percival]

> Slepian, D. Prolate spheroidal wave functions, Fourier analysis, and uncertainty V: The discrete case. Bell System Technical Journal, Volume 57 (1978), 1371430

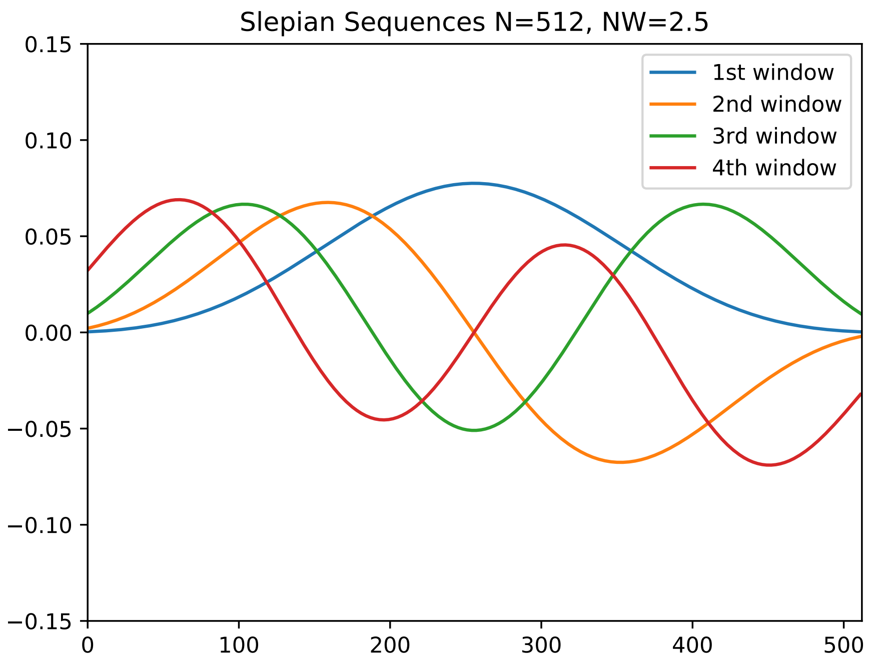
Note: the C code to create the slepian windows is extracted from original C code from Lees and Park (1995) and uses the conventions of Percival and Walden (1993). Functions that are not used here were removed.

pmtm(*x*, *NW=None*, *k=None*, *NFFT=None*, *e=None*, *v=None*, *method='adapt'*, *show=False*)

Multitapering spectral estimation

Parameters

- **x** (*array*) – the data



- **NW** (*float*) – The time half bandwidth parameter (typical values are 2.5,3,3.5,4). Must be provided otherwise the tapering windows and eigen values (outputs of dpss) must be provided
- **k** (*int*) – uses the first k Slepian sequences. If *k* is not provided, *k* is set to *NW*2*.
- **NW** –
- **e** – the window concentrations (eigenvalues)
- **v** – the matrix containing the tapering windows
- **method** (*str*) – set how the eigenvalues are used when weighting the results. Must be in ['unity', 'adapt', 'eigen']. see below for details.
- **show** (*bool*) – plot results

Returns Sk (complex), weights, eigenvalues

Usually in spectral estimation the mean to reduce bias is to use tapering window. In order to reduce variance we need to average different spectrum. The problem is that we have only one set of data. Thus we need to decompose a set into several segments. Such method are well-known: simple daniell's periodogram, Welch's method and so on. The drawback of such methods is a loss of resolution since the segments used to compute the spectrum are smaller than the data set. The interest of multitapering method is to keep a good resolution while reducing bias and variance.

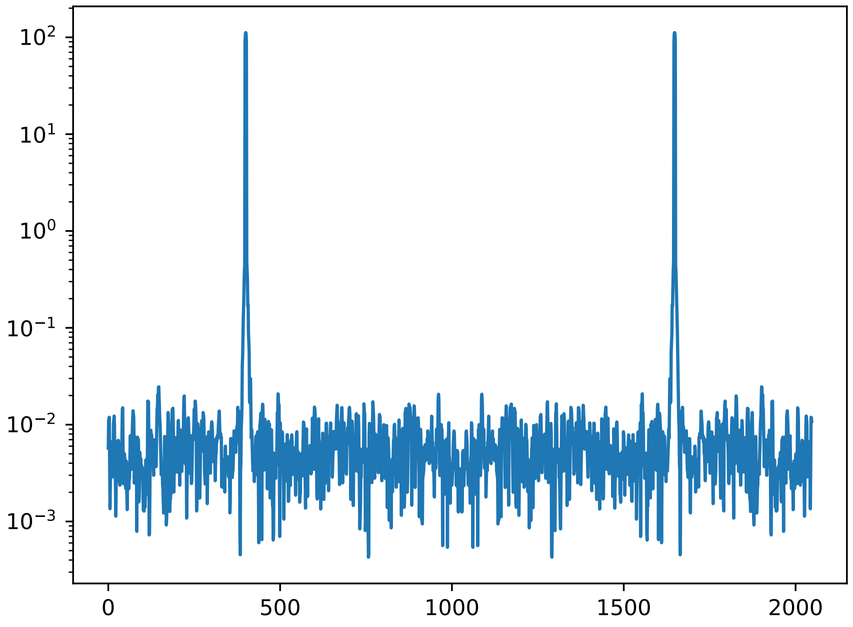
How does it work? First we compute different simple periodogram with the whole data set (to keep good resolution) but each periodgram is computed with a different tapering windows. Then, we average all these spectrum. To avoid redundancy and bias due to the tapers mtm use special tapers.

Method can be eigen, unity or adapt. If *unity*, weights are set to 1. If *eigen* are proportional to the eigen-values. If *adapt*, equations from [2] (P&W pp 368-370) are used.

The output is made of 2 matrices called *Sk* and *weights*. The third item stored the eigenvalues. The two matrices have dimensions equal to the number of windows used multiplied by the number of input points. The first matrix stored the spectral results while the second stores the weights.

Would you wish to plot the spectrum, you will have to take the means of the different windows and weight down the results before mean(Sk * weigths). Please see the code for details.

```
from spectrum import data_cosine, dpss, pmtm

data = data_cosine(N=2048, A=0.1, sampling=1024, freq=200)
# If you already have the DPSS windows
[tapers, eigen] = dpss(2048, 2.5, 4)
res = pmtm(data, e=eigen, v=tapers, show=False)
# You do not need to compute the DPSS before end
res = pmtm(data, NW=2.5, show=False)
res = pmtm(data, NW=2.5, k=4, show=True)
```



Changed in version 0.6.2: APN modified method to return each Sk as complex values, the eigenvalues and the weights

5.3 Parametric methods

Contents

5.3.1 Power Spectrum Density based on Parametric Methods

ARMA and MA estimates (yule-walker)

ARMA and MA estimates, ARMA and MA PSD estimates.

ARMA model and Power Spectral Densities.

`arma_estimate`	Autoregressive and moving average estimators.
`ma`	Moving average estimator.
`arma2psd`	Computes power spectral density given ARMA values.
`parma`	Class to create PSD using ARMA estimator.
`pma`	Class to create PSD using MA estimator.

Code author: Thomas Cokelaer 2011

References See [Marple]

arma2psd(*A=None*, *B=None*, *rho=1.0*, *T=1.0*, *NFFT=4096*, *sides='default'*, *norm=False*)

Computes power spectral density given ARMA values.

This function computes the power spectral density values given the ARMA parameters of an ARMA model. It assumes that the driving sequence is a white noise process of zero mean and variance ρ_w. The sampling frequency and noise variance are used to scale the PSD output, which length is set by the user with the *NFFT* parameter.

Parameters

- **A** (*array*) – Array of AR parameters (complex or real)
- **B** (*array*) – Array of MA parameters (complex or real)
- **rho** (*float*) – White noise variance to scale the returned PSD
- **T** (*float*) – Sampling frequency in Hertz to scale the PSD.
- **NFFT** (*int*) – Final size of the PSD
- **sides** (*str*) – Default PSD is two-sided, but sides can be set to centerdc.

Warning: By convention, the AR or MA arrays does not contain the A0=1 value.

If `B` is None, the model is a pure AR model. If `A` is None, the model is a pure MA model.

Returns two-sided PSD

Details:

AR case: the power spectral density is:

$$P_{ARMA}(f) = T\rho_w \left|\frac{B(f)}{A(f)}\right|^2$$

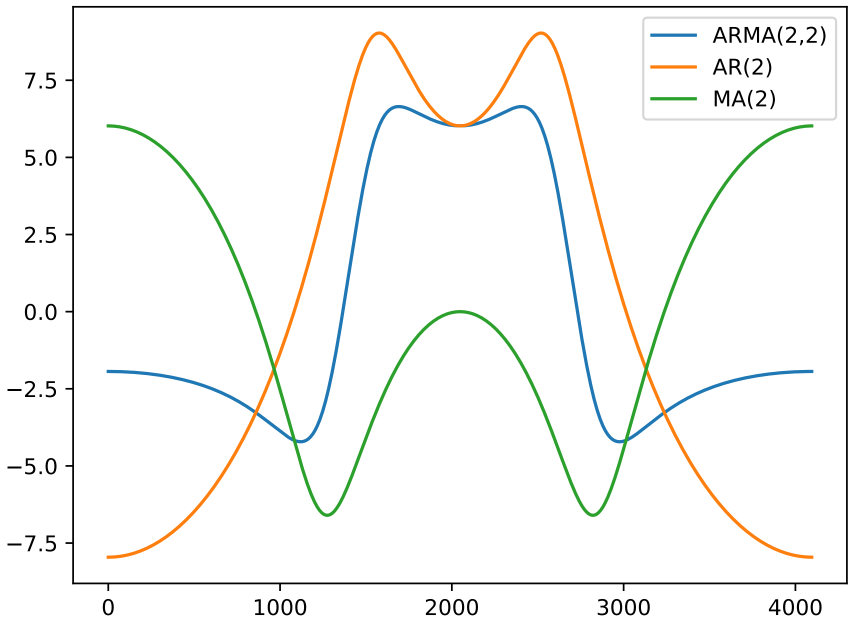
where:

$$A(f) = 1 + \sum_{k=1}^{q} b(k) e^{-j2\pi fkT}$$

$$B(f) = 1 + \sum_{k=1}^{p} a(k) e^{-j2\pi fkT}$$

Example:

```
import spectrum.arma
from pylab import plot, log10, legend
plot(10*log10(spectrum.arma.arma2psd([1,0.5],[0.5,0.5])), label='ARMA(2,2)')
plot(10*log10(spectrum.arma.arma2psd([1,0.5],None)), label='AR(2)')
plot(10*log10(spectrum.arma.arma2psd(None,[0.5,0.5])), label='MA(2)')
legend()
```



References [Marple]

arma_estimate(*X*, *P*, *Q*, *lag*)
Autoregressive and moving average estimators.

This function provides an estimate of the autoregressive parameters, the moving average parameters, and the driving white noise variance of an ARMA(P,Q) for a complex or real data sequence.

The parameters are estimated using three steps:

- Estimate the AR parameters from the original data based on a least squares modified Yule-Walker technique,
- Produce a residual time sequence by filtering the original data with a filter based on the AR parameters,
- Estimate the MA parameters from the residual time sequence.

Parameters

- **X** (*array*) – Array of data samples (length N)
- **P** (*int*) – Desired number of AR parameters
- **Q** (*int*) – Desired number of MA parameters
- **lag** (*int*) – Maximum lag to use for autocorrelation estimates

Returns

- A - Array of complex P AR parameter estimates
- B - Array of complex Q MA parameter estimates
- RHO - White noise variance estimate

Note:

- lag must be >= Q (MA order)

dependencies:

- *spectrum.correlation.CORRELATION()*
- *spectrum.covar.arcovar()*
- *spectrum.arma.ma()*

```
from spectrum import arma_estimate, arma2psd, marple_data
import pylab

a,b, rho = arma_estimate(marple_data, 15, 15, 30)
psd = arma2psd(A=a, B=b, rho=rho, sides='centerdc', norm=True)
pylab.plot(10 * pylab.log10(psd))
pylab.ylim([-50,0])
```

Reference [Marple]

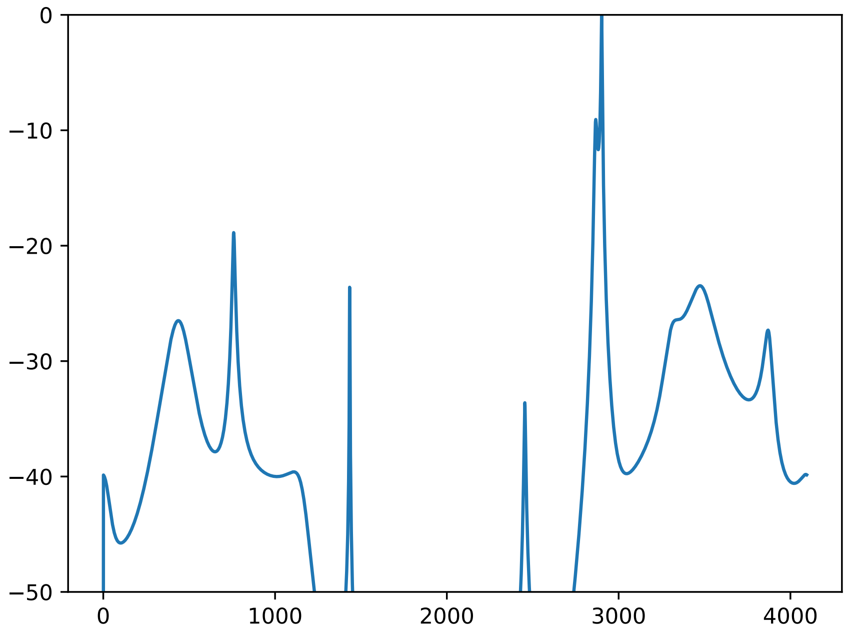
ma (*X*, *Q*, *M*)

Moving average estimator.

This program provides an estimate of the moving average parameters and driving noise variance for a data sequence based on a long AR model and a least squares fit.

Parameters

- **X** (*array*) – The input data array



- **Q** (*int*) – Desired MA model order (must be >0 and <M)
- **M** (*int*) – Order of "long" AR model (suggest at least 2*Q)

Returns

- MA - Array of Q complex MA parameter estimates
- RHO - Real scalar of white noise variance estimate

```
from spectrum import arma2psd, ma, marple_data
import pylab

# Estimate 15 Ma parameters
b, rho = ma(marple_data, 15, 30)
# Create the PSD from those MA parameters
psd = arma2psd(B=b, rho=rho, sides='centerdc')
# and finally plot the PSD
pylab.plot(pylab.linspace(-0.5, 0.5, 4096), 10 * pylab.log10(psd/max(psd)))
pylab.axis([-0.5, 0.5, -30, 0])
```

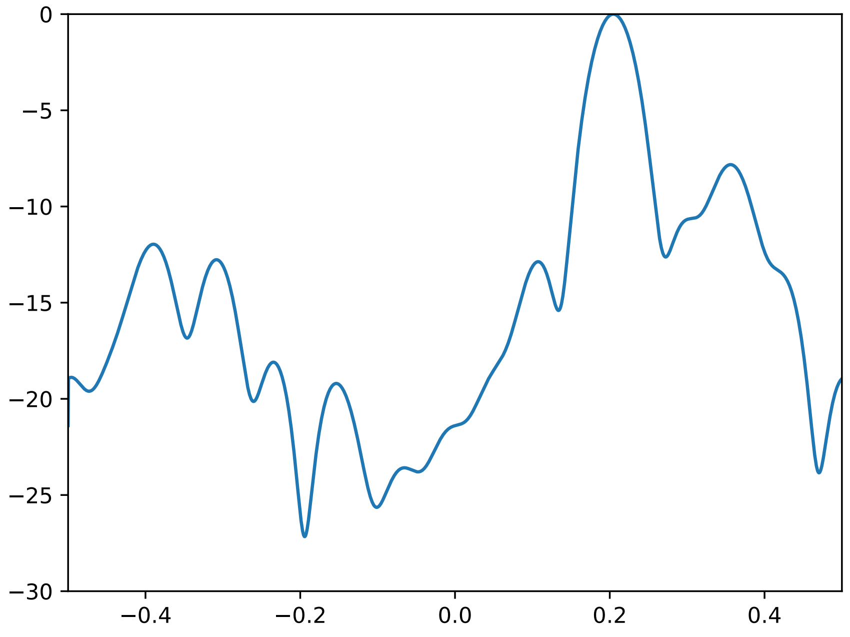
Reference [Marple]

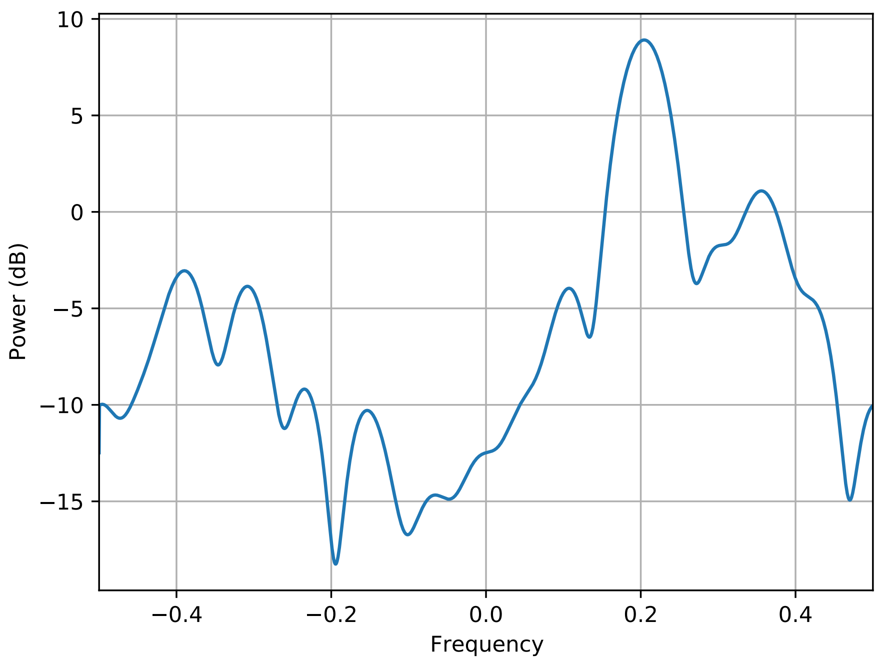
class pma(*data*, *Q*, *M*, *NFFT=None*, *sampling=1.0*, *scale_by_freq=False*)

Class to create PSD using MA estimator.

See *ma()* for description.

```
from spectrum import pma, marple_data
p = pma(marple_data, 15, 30, NFFT=4096)
p.plot(sides='centerdc')
```


0
−5
−10
−15
−20
−25
−30
−0.4
−0.2
0.0
0.2
0.4


10
5
0
−5
−10
−15
Power (dB)
−0.4
−0.2
0.0
0.2
0.4
Frequency

Constructor:

For a detailed description of the parameters, see `ma()`.

Parameters

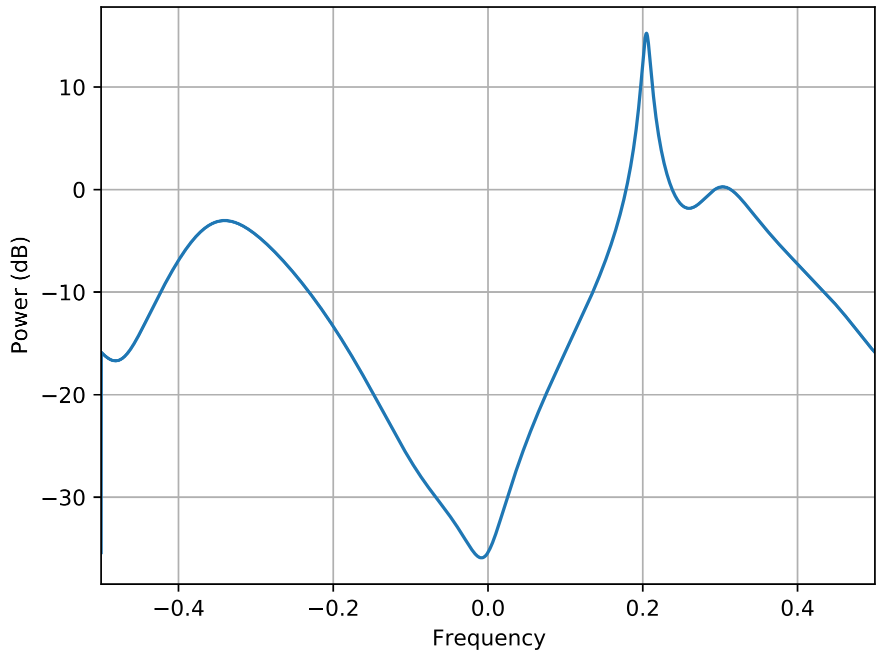
- **data** (*array*) – input data (list or numpy.array)
- **Q** (*int*) – MA order
- **M** (*int*) – AR model used to estimate the MA parameters
- **NFFT** (*int*) – total length of the final data sets (padded with zero if needed; default is 4096)
- **sampling** (*float*) – sampling frequency of the input `data`.

class parma (*data*, *P*, *Q*, *lag*, *NFFT=None*, *sampling=1.0*, *scale_by_freq=False*)

Class to create PSD using ARMA estimator.

See `arma_estimate()` for description.

```
from spectrum import parma, marple_data
p = parma(marple_data, 4, 4, 30, NFFT=4096)
p.plot(sides='centerdc')
```



Constructor:

For a detailed description of the parameters, see `arma_estimate()`.

Parameters

- **data** (*array*) – input data (list or numpy.array)
- **P** (*int*) –
- **Q** (*int*) –

- **lag** (*int*) –
- **NFFT** (*int*) – total length of the final data sets (padded with zero if needed; default is 4096)
- **sampling** (*float*) – sampling frequency of the input `data`.

AR estimate based on Burg algorithm

BURG method of AR model estimate

This module provides BURG method and BURG PSD estimate

arburg(X, order[, criteria])	Estimate the complex autoregressive parameters by the Burg algorithm.
pburg(data, order[, criteria, NFFT, ...])	Class to create PSD based on Burg algorithm

Code author: Thomas Cokelaer 2011

arburg (*X*, *order*, *criteria=None*)

Estimate the complex autoregressive parameters by the Burg algorithm.

$$x(n) = \sqrt{(v)}e(n) + \sum_{k=1}^{P+1} a(k)x(n-k)$$

Parameters

- **x** – Array of complex data samples (length N)
- **order** – Order of autoregressive process (0<order<N)
- **criteria** – select a criteria to automatically select the order

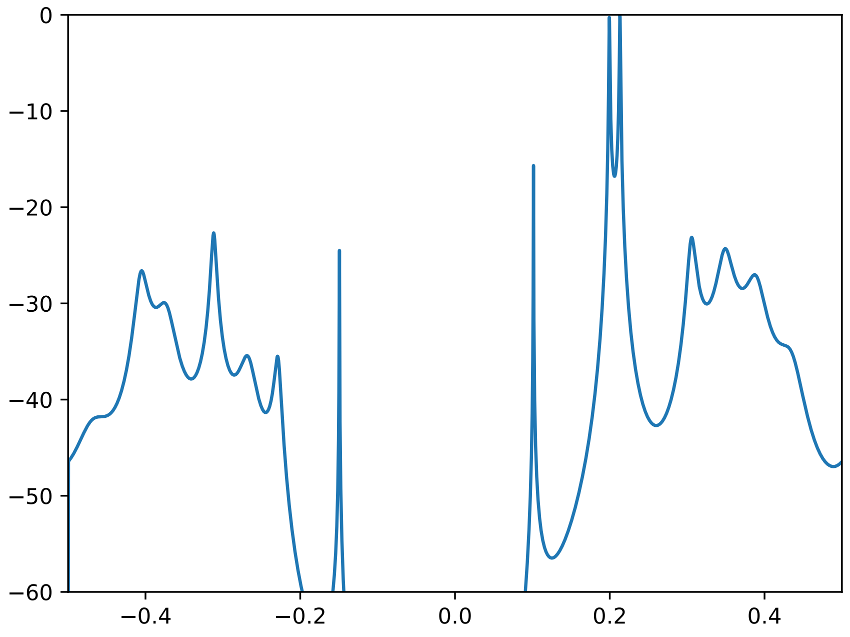
Returns

- A Array of complex autoregressive parameters A(1) to A(order). First value (unity) is not included !!
- P Real variable representing driving noise variance (mean square of residual noise) from the whitening operation of the Burg filter.
- reflection coefficients defining the filter of the model.

```
from pylab import plot, log10, linspace, axis
from spectrum import *

AR, P, k = arburg(marple_data, 15)
PSD = arma2psd(AR, sides='centerdc')
plot(linspace(-0.5, 0.5, len(PSD)), 10*log10(PSD/max(PSD)))
axis([-0.5,0.5,-60,0])
```

Note:

1. no detrend. Should remove the mean trend to get PSD. Be careful if presence of large mean.



2. If you don't know what the order value should be, choose the criterion='AKICc', which has the least bias and best resolution of model-selection criteria.

Note: real and complex results double-checked versus octave using complex 64 samples stored in marple_data. It does not agree with Marple fortran routine but this is due to the simplex precision of complex data in fortran.

Reference [Marple] [octave]

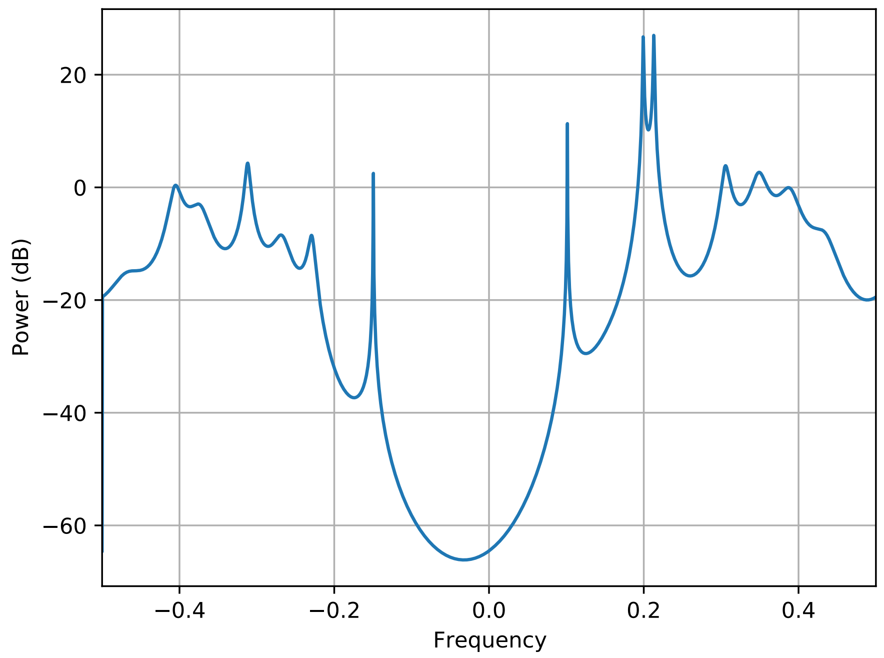
class pburg(*data*, *order*, *criteria=None*, *NFFT=None*, *sampling=1.0*, *scale_by_freq=False*)
Class to create PSD based on Burg algorithm

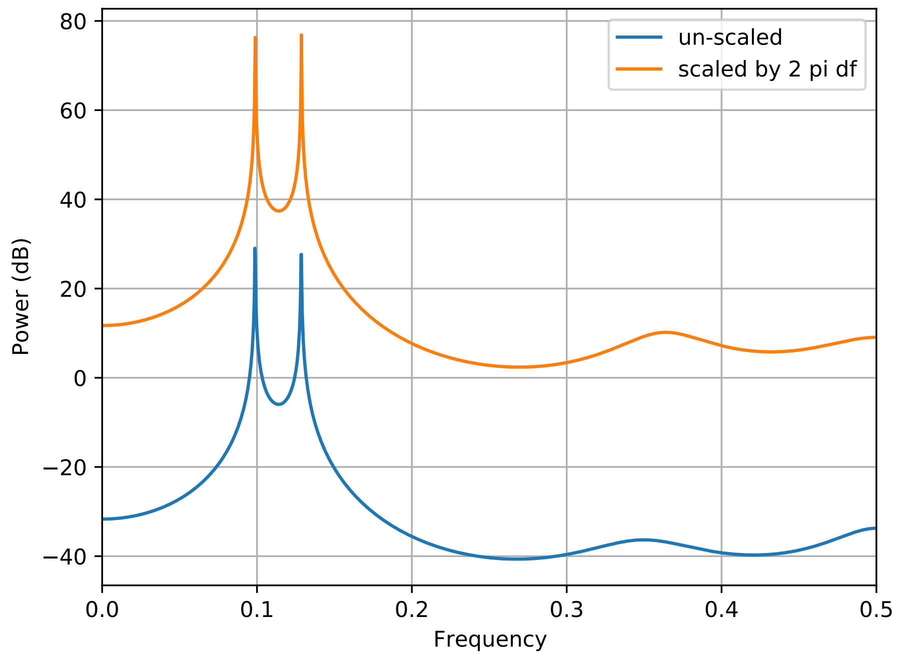
See *arburg()* for description.

```
from spectrum import *
p = pburg(marple_data, 15, NFFT=4096)
p.plot(sides='centerdc')
```

Another example based on a real data set is shown here below. Note here that we set the scale_by_freq value to False and True. False should give results equivalent to octave or matlab convention while setting to True implies that the data is multiplied by $2\pi df$ where $df = sampling/N$.

```
from spectrum import data_two_freqs, pburg
p = pburg(data_two_freqs(), 7, NFFT=4096)
p.plot()
p = pburg(data_two_freqs(), 7, NFFT=4096, scale_by_freq=True)
p.plot()
from pylab import legend
legend(["un-scaled", "scaled by 2 pi df"])
```


20
0
−20
−40
−60
Power (dB)
−0.4
−0.2
0.0
0.2
0.4
Frequency


un-scaled
scaled by 2 pi df
80
60
40
20
0
−20
−40
Power (dB)
0.0
0.1
0.2
0.3
0.4
0.5
Frequency

Constructor

For a detailled description of the parameters, see `burg()`.

Parameters

- **`data`** (*`array`*) – input data (list or np.array)
- **`order`** (*`int`*) –
- **`criteria`** (*`str`*) –
- **`NFFT`** (*`int`*) – total length of the final data sets (padded with zero if needed; default is 4096)
- **`sampling`** (*`float`*) – sampling frequency of the input `data`.

AR estimate based on YuleWalker

Yule Walker method to estimate AR values.

Estimation of AR values using Yule-Walker method

`aryule`(X, order[, norm, allow_singularity])	Compute AR coefficients using Yule-Walker method
`pyule`(data, order[, norm, NFFT, sampling, . . .])	Class to create PSD based on the Yule Walker method

Code author: Thomas Cokelaer 2011

`aryule` (*X*, *order*, *norm='biased'*, *allow_singularity=True*)

Compute AR coefficients using Yule-Walker method

Parameters

- **`X`** – Array of complex data values, X(1) to X(N)
- **`order`** (*`int`*) – Order of autoregressive process to be fitted (integer)
- **`norm`** (*`str`*) – Use a biased or unbiased correlation.
- **`allow_singularity`** (*`bool`*) –

Returns

- AR coefficients (complex)
- variance of white noise (Real)
- reflection coefficients for use in lattice filter

Description:

The Yule-Walker method returns the polynomial A corresponding to the AR parametric signal model estimate of vector X using the Yule-Walker (autocorrelation) method. The autocorrelation may be computed using a **biased** or **unbiased** estimation. In practice, the biased estimate of the autocorrelation is used for the unknown true autocorrelation. Indeed, an unbiased estimate may result in nonpositive-definite autocorrelation matrix. So, a biased estimate leads to a stable AR filter. The following matrix form represents the Yule-Walker equations. The are solved by means of the Levinson-Durbin recursion:

$$\begin{pmatrix} r(1) & r(2)^* & \dots & r(n)^* \\ r(2) & r(1)^* & \dots & r(n-1)^* \\ \dots & \dots & \dots & \dots \\ r(n) & \dots & r(2) & r(1) \end{pmatrix} \begin{pmatrix} a(2) \\ a(3) \\ \dots \\ a(n+1) \end{pmatrix} = \begin{pmatrix} -r(2) \\ -r(3) \\ \dots \\ -r(n+1) \end{pmatrix}$$

The outputs consists of the AR coefficients, the estimated variance of the white noise process, and the reflection coefficients. These outputs can be used to estimate the optimal order by using `criteria`.

Examples:

From a known AR process or order 4, we estimate those AR parameters using the aryule function.

```
>>> from scipy.signal import lfilter
>>> from spectrum import *
>>> from numpy.random import randn
>>> A  =[1, -2.7607, 3.8106, -2.6535, 0.9238]
>>> noise = randn(1, 1024)
>>> y = lfilter([1], A, noise);
>>> #filter a white noise input to create AR(4) process
>>> [ar, var, reflec] = aryule(y[0], 4)
>>> # ar should contains values similar to A
```

The PSD estimate of a data samples is computed and plotted as follows:

```
from spectrum import *
from pylab import *

ar, P, k = aryule(marple_data, 15, norm='biased')
psd = arma2psd(ar)
plot(linspace(-0.5, 0.5, 4096), 10 * log10(psd/max(psd)))
axis([-0.5, 0.5, -60, 0])
```

Note: The outputs have been double checked against (1) octave outputs (octave has norm='biased' by default) and (2) Marple test code.

See also:

This function uses `LEVINSON()` and `CORRELATION()`. See the `criteria` module for criteria to automatically select the AR order.

References [Marple]

class pyule(*data*, *order*, *norm='biased'*, *NFFT=None*, *sampling=1.0*, *scale_by_freq=True*)

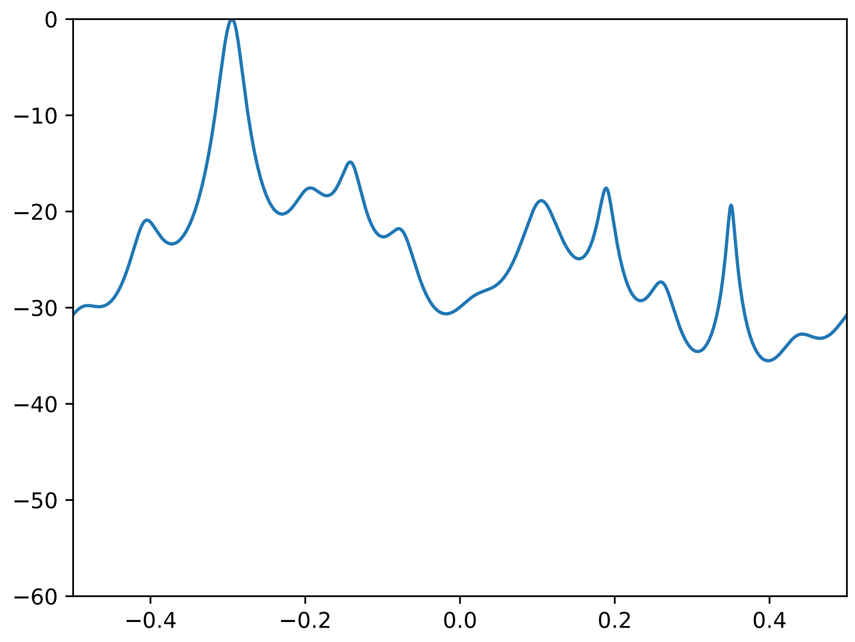
Class to create PSD based on the Yule Walker method

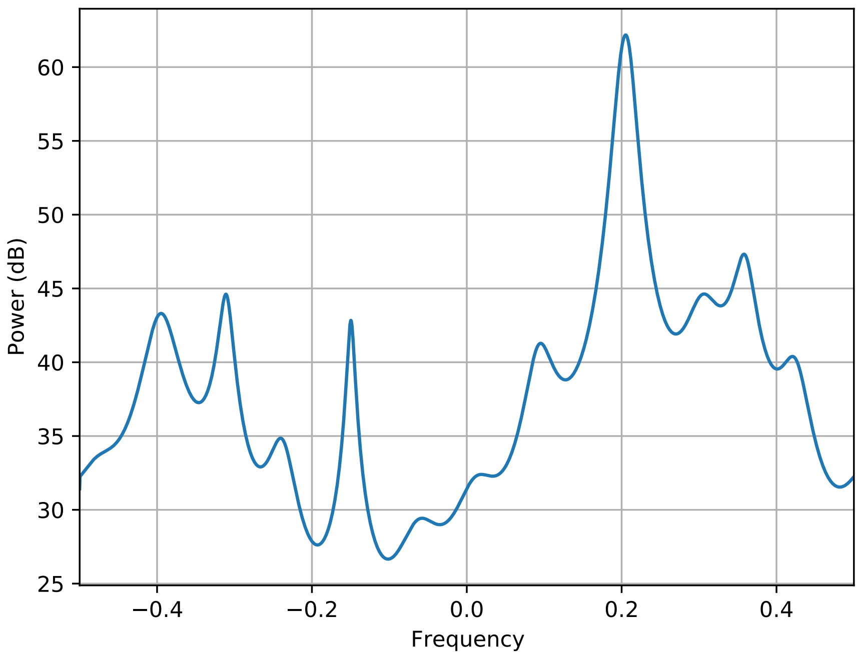
See `aryule()` for description.

```
from spectrum import *
p = pyule(marple_data, 15, NFFT=4096)
p.plot(sides='centerdc')
```

Constructor

For a detailled description of the parameters, see `aryule()`.


0
−10
−20
−30
−40
−50
−60
−0.4
−0.2
0.0
0.2
0.4


60
55
50
45
40
35
30
25
Power (dB)
−0.4
−0.2
0.0
0.2
0.4
Frequency

Parameters

- **data** (*array*) – input data (list or numpy.array)
- **order** (*int*) –
- **NFFT** (*int*) – total length of the final data sets (padded with zero if needed; default is 4096)
- **sampling** (*float*) – sampling frequency of the input `data`
- **norm** (*str*) – don't change if you do not know

5.3.2 Criteria

Criteria for parametric methods.

This module provides criteria to automatically select order in parametric PSD estimate or pseudo spectrum estimates (e.g, music).

Some criteria such as the AIC criterion helps to chose the order of PSD models such as the ARMA model. Nevertheless, it is difficult to estimate correctly the order of an ARMA model even by using these criteria. The reason being that even the Akaike criteria (AIC) does not provide the proper order with a probability of 1 with infinite samples.

The order choice is related to an expertise of the signal. There is no exact criteria. However, they may provide useful information.

AIC, AICc, KIC and AKICc are based on information theory. They attempt to balance the complexity (or length) of the model against how well the model fits the data. AIC and KIC are biased estimates of the asymmetric and the symmetric Kullback-Leibler divergence respectively. AICc and AKICc attempt to correct the bias.

There are also criteria related to eigen analysis, which takes as input the eigen values of any PSD estimate method.

Example

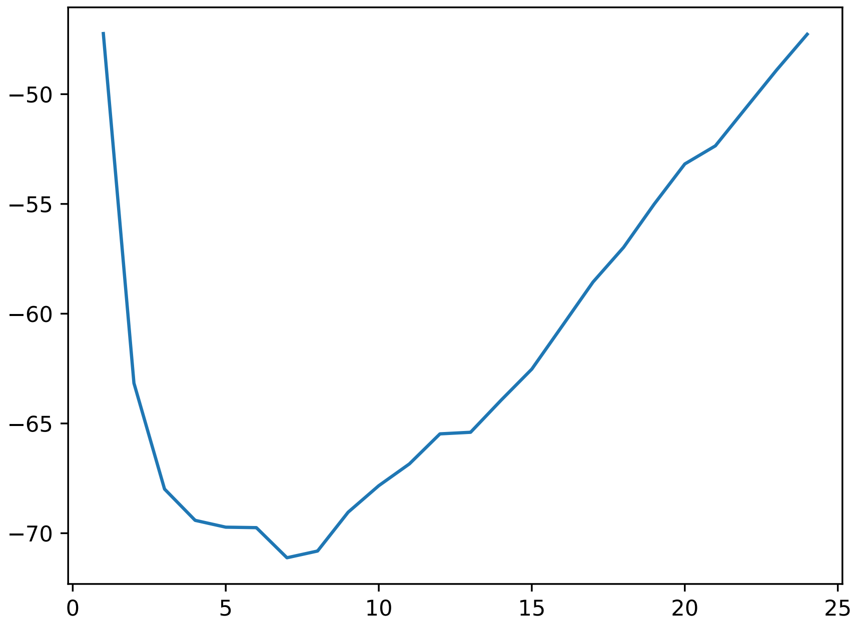
```
from spectrum import aryule, AIC, marple_data
from pylab import plot, arange
order = arange(1, 25)
rho = [aryule(marple_data, i, norm='biased')[1] for i in order]
plot(order, AIC(len(marple_data), rho, order), label='AIC')
```

References bd-Krim Seghouane and Maiza Bekara "A small sample model selection criterion based on Kullback's symmetric divergence", IEEE Transactions on Signal Processing, Vol. 52(12), pp 3314-3323, Dec. 2004

AIC (*N*, *rho*, *k*)
Akaike Information Criterion

Parameters

- **rho** – rho at order k
- **N** – sample size
- **k** – AR order.



If k is the AR order and N the size of the sample, then Akaike criterion is

$$AIC(k) = \log(\rho_k) + 2\frac{k+1}{N}$$

```
AIC(64, [0.5,0.3,0.2], [1,2,3])
```

Validation double checked versus octave.

AICc (*N*, *rho*, *k*, *norm=True*)
corrected Akaike information criterion

$$AICc(k) = log(\rho_k) + 2\frac{k+1}{N-k-2}$$

Validation double checked versus octave.

AKICc (*N*, *rho*, *k*)
approximate corrected Kullback information

$$AKICc(k) = log(rho_k) + \frac{p}{N*(N-k)} + (3-\frac{k+2}{N})*\frac{k+1}{N-k-2}$$

CAT (*N*, *rho*, *k*)
Criterion Autoregressive Transfer Function :

$$CAT(k) = \frac{1}{N}\sum_{i=1}^{k}\frac{1}{\rho_i} - \frac{\rho_i}{\rho_k}$$

Todo: validation

class Criteria(*name*, *N*)

Criteria class for an automatic selection of ARMA order.

Available criteria are

AIC	see *AIC()*
AICc	see *AICc()*
KIC	see *KIC()*
AKICc	see *AKICc()*
FPE	see *FPE()*
MDL	see *MDL()*
CAT	see _CAT()

Create a criteria object

Parameters

- **name** – a string or list of strings containing valid criteria method's name
- **N** (*int*) – size of the data sample.

N

Getter/Setter for N

data

Getter/Setter for the criteria output

error_incorrect_name = "Invalid name provided. Correct names are ['AIC', 'AICc', 'KIC'

error_no_criteria_found = "No names match the valid criteria names (['AIC', 'AICc', 'K

k

Getter for k the order of evaluation

name

Getter/Setter for the criteria name

old_data

Getter/Setter for the previous value

rho

Getter/Setter for rho

valid_criteria_names = ['AIC', 'AICc', 'KIC', 'FPE', 'AKICc', 'MDL']

FPE(*N*, *rho*, *k=None*)

Final prediction error criterion

$$FPE(k) = \frac{N+k+1}{N-k-1}\rho_k$$

Validation double checked versus octave.

KIC(*N*, *rho*, *k*)

Kullback information criterion

$$KIC(k) = log(\rho_k) + 3\frac{k+1}{N}$$

Validation double checked versus octave.

MDL(*N*, *rho*, *k*)
Minimum Description Length

$$MDL(k) = N log \rho_k + p \log N$$

Validation results

aic_eigen(*s*, *N*)
AIC order-selection using eigen values

Parameters

- **s** – a list of p sorted eigen values
- **N** – the size of the input data. To be defined precisely.

Returns

- an array containing the AIC values

Given n sorted eigen values λ_i with $0 <= i < n$, the proposed criterion from Wax and Kailath (1985) is:

$$AIC(k) = -2(n-k)N \ln \frac{g(k)}{a(k)} + 2k(2n-k)$$

where the arithmetic sum $a(k)$ is:

$$a(k) = \sum_{i=k+1}^{n} \lambda_i$$

and the geometric sum $g(k)$ is:

$$g(k) = \prod_{i=k+1}^{n} \lambda_i^{-(n-k)}$$

The number of relevant sinusoids in the signal subspace is determined by selecting the minimum of *AIC*.

See also:

`eigen()`

Todo: define precisely the input parameter N. Should be the input data length but when using correlation matrix (SVD), I suspect it should be the length of the correlation matrix rather than the original data.

References

- [Marple] Chap 13,
- [Wax]

mdl_eigen(*s*, *N*)
MDL order-selection using eigen values

Parameters

- **s** – a list of p sorted eigen values
- **N** – the size of the input data. To be defined precisely.

Returns

- an array containing the AIC values

$$MDL(k) = (n-k)N \ln \frac{g(k)}{a(k)} + 0.5k(2n-k)log(N)$$

See also:

`aic_eigen()` for details

References

- [Marple] Chap 13,
- [Wax]

5.4 Other Power Spectral Density estimates

Contents

5.4.1 Covariance method

arcovar (*x*, *order*)

Simple and fast implementation of the covariance AR estimate

This code is 10 times faster than `arcovar_marple()` and more importantly only 10 lines of code, compared to a 200 loc for `arcovar_marple()`

Parameters

- **X** (*array*) – Array of complex data samples
- **oder** (*int*) – Order of linear prediction model

Returns

- a - Array of complex forward linear prediction coefficients
- e - error

The covariance method fits a Pth order autoregressive (AR) model to the input signal, which is assumed to be the output of an AR system driven by white noise. This method minimizes the forward prediction error in the least-squares sense. The output vector contains the normalized estimate of the AR system parameters

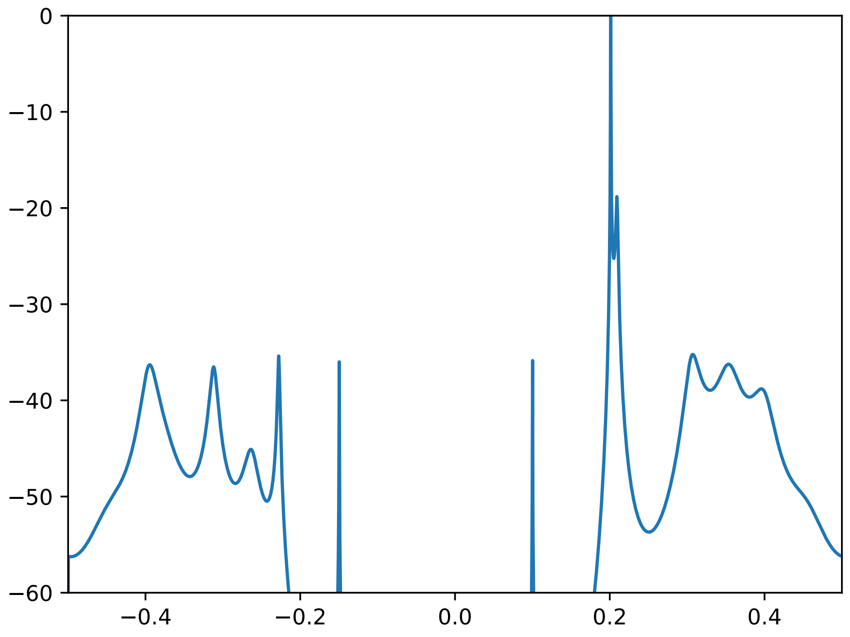
The white noise input variance estimate is also returned.

If is the power spectral density of y(n), then:

$$\frac{e}{\left|A(e^{jw})\right|^2} = \frac{e}{\left|1+\sum_{k-1}^P a(k)e^{-jwk}\right|^2}$$

Because the method characterizes the input data using an all-pole model, the correct choice of the model order p is important.

```
from spectrum import arcovar, marple_data, arma2psd
from pylab import plot, log10, linspace, axis

ar_values, error = arcovar(marple_data, 15)
psd = arma2psd(ar_values, sides='centerdc')
plot(linspace(-0.5, 0.5, len(psd)), 10*log10(psd/max(psd)))
axis([-0.5, 0.5, -60, 0])
```



See also:

pcovar

Validation the AR parameters are the same as those returned by a completely different function *arcovar_marple()*.

References [Mathworks]

arcovar_marple(*x*, *order*)

Estimate AR model parameters using covariance method

This implementation is based on [Marple]. This code is far more complicated and slower than *arcovar()* function, which is now the official version. See *arcovar()* for a detailed description of Covariance method.

This function should be used in place of arcovar only if order<=4, for which *arcovar()* does not work.

Fast algorithm for the solution of the covariance least squares normal equations from Marple.

Parameters

- **X** (*array*) – Array of complex data samples
- **oder** (*int*) – Order of linear prediction model

Returns

- AF - Array of complex forward linear prediction coefficients
- PF - Real forward linear prediction variance at order IP
- AB - Array of complex backward linear prediction coefficients
- PB - Real backward linear prediction variance at order IP
- PV - store linear prediction coefficients

Note: this code and the original code in Marple diverge for ip>10. it seems that this is related to single precision used with complex type in fortran whereas numpy uses double precision for complex type.

Validation the AR parameters are the same as those returned by a completely different function `arcovar()`.

References [Marple]

class pcovar(*data*, *order*, *NFFT=None*, *sampling=1.0*, *scale_by_freq=False*)

Class to create PSD based on covariance algorithm

See `arcovar()` for description.

```
from spectrum import pcovar, marple_data
p = pcovar(marple_data, 15, NFFT=4096)
p.plot(sides='centerdc')
```

See also:

`arcovar`

Constructor

For a detailled description of the parameters, see `arcovar()`.

Parameters

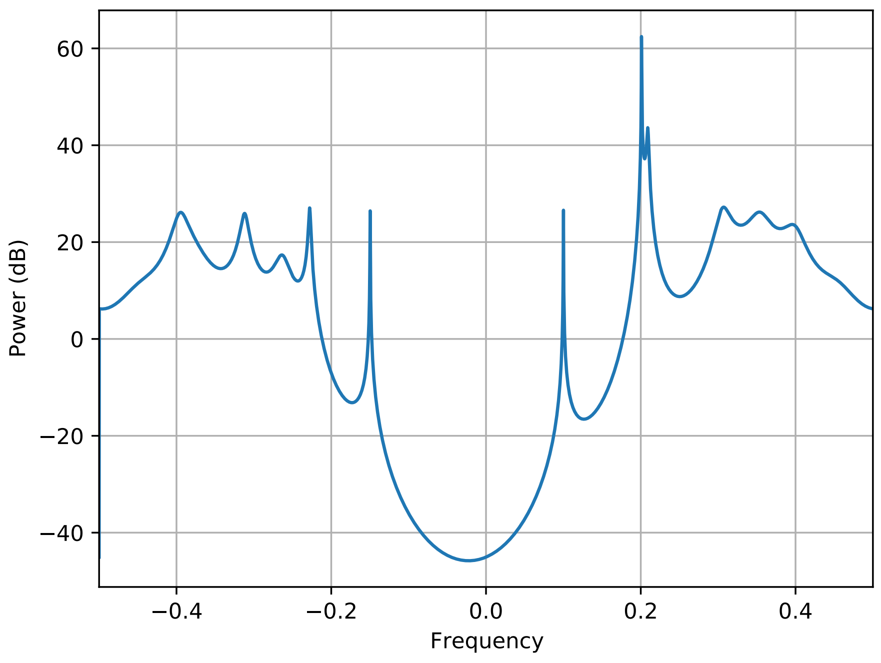
- **data** (*array*) – input data (list or numpy.array)
- **order** (*int*) –
- **NFFT** (*int*) – total length of the final data sets (padded with zero if needed; default is 4096)
- **sampling** (*float*) – sampling frequency of the input `data`.

arcovar_marple(*x*, *order*)

Estimate AR model parameters using covariance method

This implementation is based on [Marple]. This code is far more complicated and slower than `arcovar()` function, which is now the official version. See `arcovar()` for a detailed description of Covariance method.

This function should be used in place of arcovar only if order<=4, for which `arcovar()` does not work.



Fast algorithm for the solution of the covariance least squares normal equations from Marple.

Parameters

- **X** (*array*) – Array of complex data samples
- **oder** (*int*) – Order of linear prediction model

Returns

- AF - Array of complex forward linear prediction coefficients
- PF - Real forward linear prediction variance at order IP
- AB - Array of complex backward linear prediction coefficients
- PB - Real backward linear prediction variance at order IP
- PV - store linear prediction coefficients

Note: this code and the original code in Marple diverge for ip>10. it seems that this is related to single precision used with complex type in fortran whereas numpy uses double precision for complex type.

Validation the AR parameters are the same as those returned by a completely different function `arcovar()`.

References [Marple]

5.4.2 Eigen-analysis methods

eigen (*X*, *P*, *NSIG=None*, *method='music'*, *threshold=None*, *NFFT=4096*, *criteria='aic'*, *verbose=False*)
Pseudo spectrum using eigenvector method (EV or Music)

This function computes either the Music or EigenValue (EV) noise subspace frequency estimator.

First, an autocorrelation matrix of order *P* is computed from the data. Second, this matrix is separated into vector subspaces, one a signal subspace and the other a noise subspace using a SVD method to obtain the eigen values and vectors. From the eigen values λ_i, and eigen vectors v_k, the **pseudo spectrum** (see note below) is computed as follows:

$$P_{ev}(f) = \frac{1}{e^H(f)\left(\sum\limits_{k=M+1}^{p} \frac{1}{\lambda_k} v_k v_k^H\right) e(f)}$$

The separation of the noise and signal subspaces requires expertise of the signal. However, AIC and MDL criteria may be used to automatically perform this task.

You still need to provide the parameter *P* to indicate the maximum number of eigen values to be computed. The criteria will just select a subset to estimate the pseudo spectrum (see `aic_eigen()` and `mdl_eigen()` for details.

Note: pseudo spectrum. func:*eigen* does not compute a PSD estimate. Indeed, the method does not preserve the measured process power.

Parameters

- **X** – Array data samples
- **P** (*int*) – maximum number of eigen values to compute. NSIG (if specified) must therefore be less than P.
- **method** (*str*) – 'music' or 'ev'.
- **NSIG** (*int*) – If specified, the signal sub space uses NSIG eigen values.
- **threshold** (*float*) – If specified, the signal sub space is made of the eigen values larger than $\text{threshold} \times \lambda_{\text{min}}$, where λ_{min} is the minimum eigen values.
- **NFFT** (*int*) – total length of the final data sets (padded with zero if needed; default is 4096)

Returns

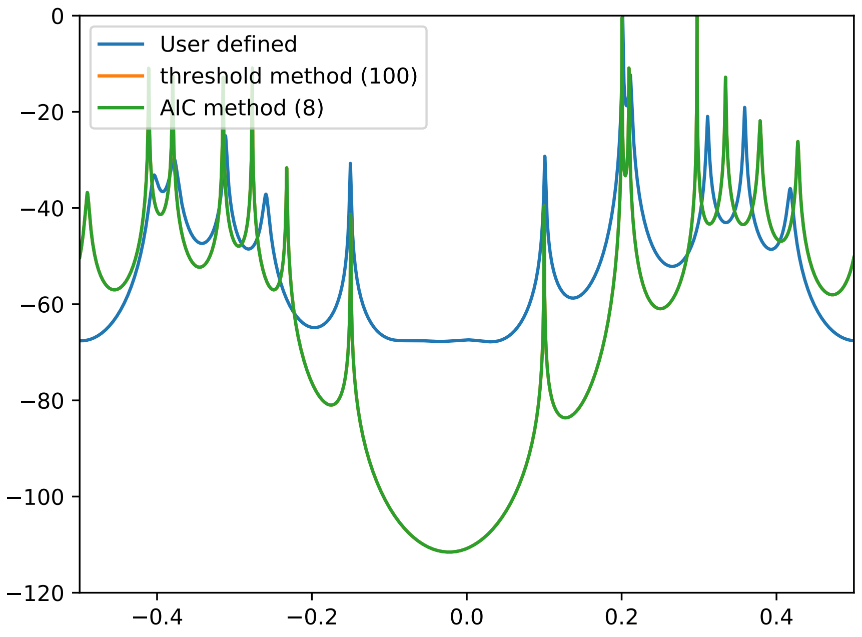
- **PSD: Array of real frequency estimator values (two sided for** complex data and one sided for real data)
- S, the eigen values

```
from spectrum import eigen, marple_data
from pylab import plot, log10, linspace, legend, axis

psd, ev = eigen(marple_data, 15, NSIG=11)
f = linspace(-0.5, 0.5, len(psd))
plot(f, 10 * log10(psd/max(psd)), label='User defined')

psd, ev = eigen(marple_data, 15, threshold=2)
```

(continues on next page)

(continued from previous page)

```
plot(f, 10 * log10(psd/max(psd)), label='threshold method (100)')

psd, ev = eigen(marple_data, 15)
plot(f, 10 * log10(psd/max(psd)), label='AIC method (8)')

legend()
axis([-0.5, 0.5, -120, 0])
```



See also:

pev(), *pmusic()*, *aic_eigen()*

References [Marple], Chap 13

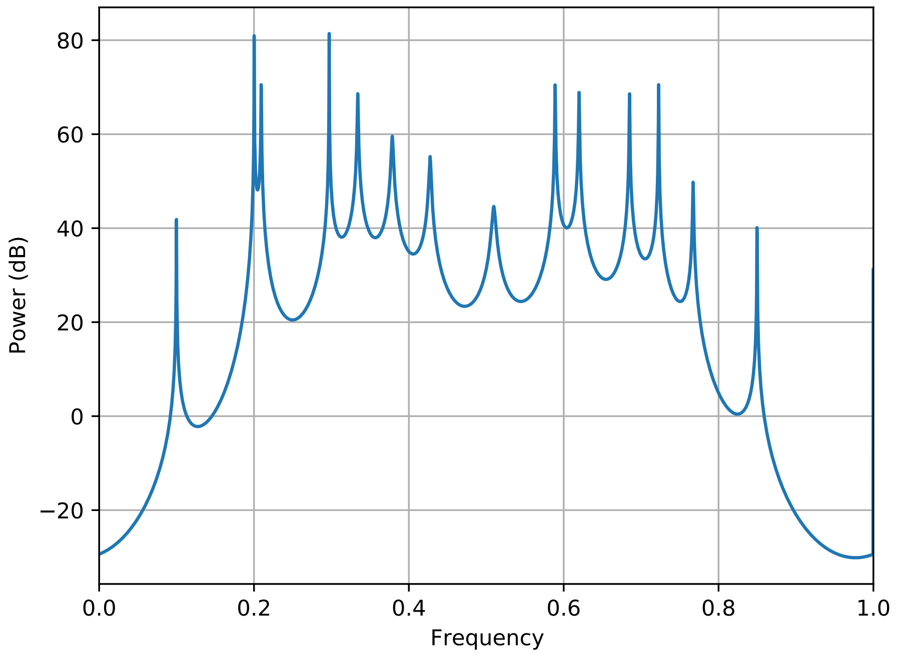
Todo: for developers:

- what should be the second argument of the criteria N, N-P, P. . . ?
- what should be the max value of NP

class pev(*data*, *IP*, *NSIG=None*, *NFFT=None*, *sampling=1.0*, *scale_by_freq=False*, *threshold=None*, *criteria='aic'*, *verbose=False*)
Class to create PSD using ARMA estimator.

See *eigenfre()* for description.

```
from spectrum import pev, marple_data
p = pev(marple_data, 15, NFFT=4096)
p.plot()
```



Constructor:

For a detailed description of the parameters, see `arma_estimate()`.

Parameters

- **`data`** (*`array`*) – input data (list or numpy.array)
- **`P`** (*`int`*) – maximum number of eigen values to compute. NSIG (if specified) must therefore be less than P.
- **`NSIG`** (*`int`*) – If specified, the signal sub space uses NSIG eigen values.
- **`NFFT`** (*`int`*) – total length of the final data sets (padded with zero if needed; default is 4096)
- **`sampling`** (*`float`*) – sampling frequency of the input `data`.

`class pmusic`(*data*, *IP*, *NSIG=None*, *NFFT=None*, *sampling=1.0*, *threshold=None*, *criteria='aic'*, *verbose=False*, *scale_by_freq=False*)

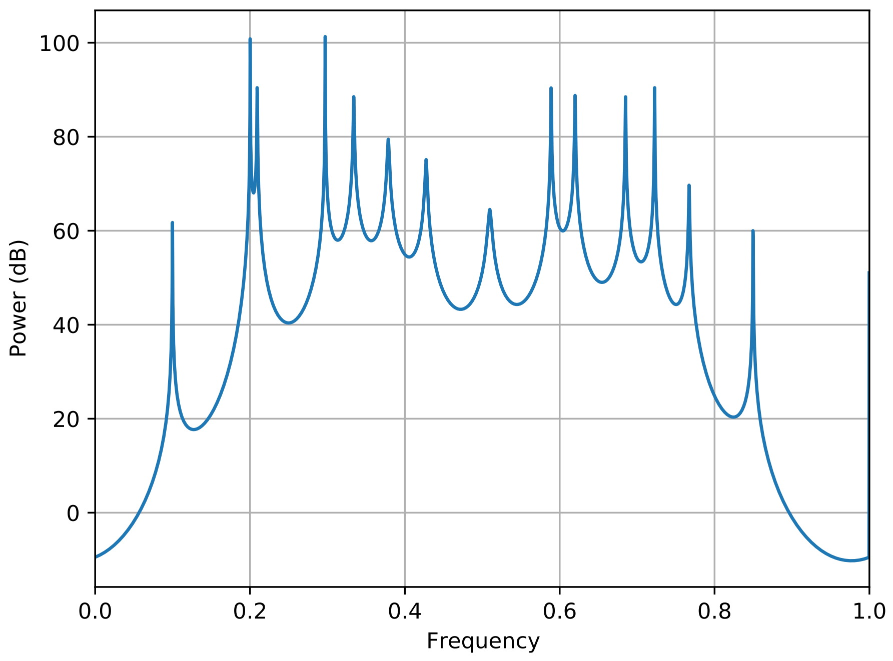
Class to create PSD using ARMA estimator.

See *`pmusic()`* and *`eigenfre()`* for description.

```
from spectrum import pmusic, marple_data
p = pmusic(marple_data, 15, NFFT=4096)
p.plot()
```

Another example using a two sinusoidal components:

```
n = arange(200)
x = cos(0.257*pi*n) + sin(0.2*pi*n) + 0.01*randn(size(n));
p = pmusic(x, 6,4)
p.plot()
```



Constructor:

For a detailed description of the parameters, see `arma_estimate()`.

Parameters

- **`data`** (*`array`*) – input data (list or numpy.array)
- **`P`** (*`int`*) – maximum number of eigen values to compute. NSIG (if specified) must therefore be less than P.
- **`NSIG`** (*`int`*) – If specified, the signal sub space uses NSIG eigen values.
- **`NFFT`** (*`int`*) – total length of the final data sets (padded with zero if needed; default is 4096)
- **`sampling`** (*`float`*) – sampling frequency of the input `data`.

5.4.3 Minimum Variance estimator

Minimum Variance Spectral Estimators

`pminvar`(data, order[, NFFT, sampling, . . .])	Class to create PSD based on the Minimum variance spectral estimation

Code author: Thomas Cokelaer, 2011

`minvar` (*X*, *order*, *sampling=1.0*, *NFFT=4096*)

Minimum Variance Spectral Estimation (MV)

This function computes the minimum variance spectral estimate using the Musicus procedure. The Burg algorithm from `arburg()` is used for the estimation of the autoregressive parameters. The MV spectral estimator is given by:

$$P_{MV}(f) = \frac{T}{e^H(f)R_p^{-1}e(f)}$$

where R_p^{-1} is the inverse of the estimated autocorrelation matrix (Toeplitz) and $e(f)$ is the complex sinusoid vector.

Parameters

- **X** – Array of complex or real data samples (length N)
- **order** (*int*) – Dimension of correlation matrix (AR order = order - 1)
- **T** (*float*) – Sample interval (PSD scaling)
- **NFFT** (*int*) – length of the final PSD

Returns

- PSD - Power spectral density values (two-sided)
- AR - AR coefficients (Burg algorithm)
- k - Reflection coefficients (Burg algorithm)

Note: The MV spectral estimator is not a true PSD function because the area under the MV estimate does not represent the total power in the measured process. MV minimises the variance of the output of a narrowband filter and adpats itself to the spectral content of the input data at each frequency.

Example The following example computes a PSD estimate using `minvar()` The output PSD is transformed to a `centerdc` PSD and plotted.

```
from spectrum import *
from pylab import plot, log10, linspace, xlim
psd, A, k = minvar(marple_data, 15)
psd = twosided_2_centerdc(psd) # switch positive and negative freq
f = linspace(-0.5, 0.5, len(psd))
plot(f, 10 * log10(psd/max(psd)))
xlim(-0.5, 0.5 )
```

See also:

- External functions used are `arburg()` and numpy.fft.fft
- `pminvar`, a Class dedicated to MV method.

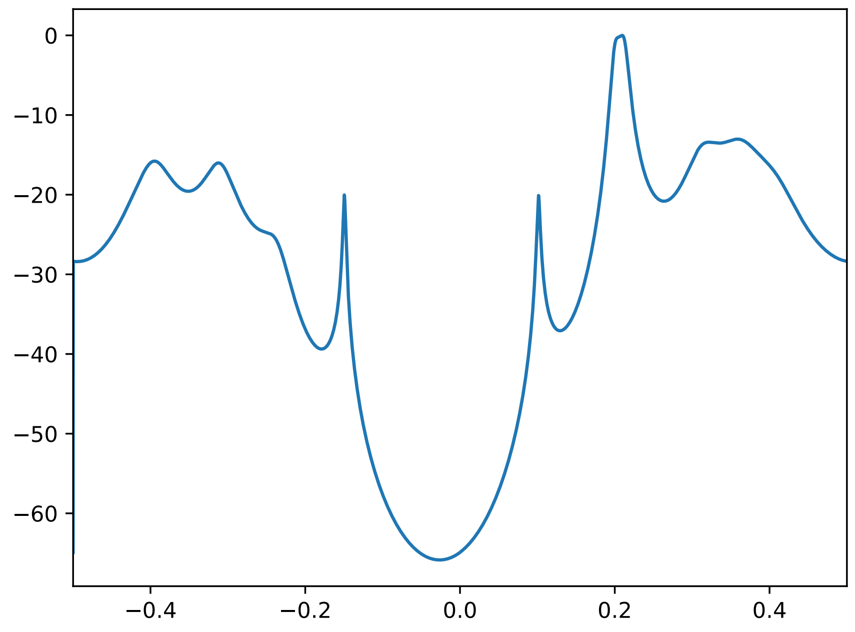
Reference [Marple]

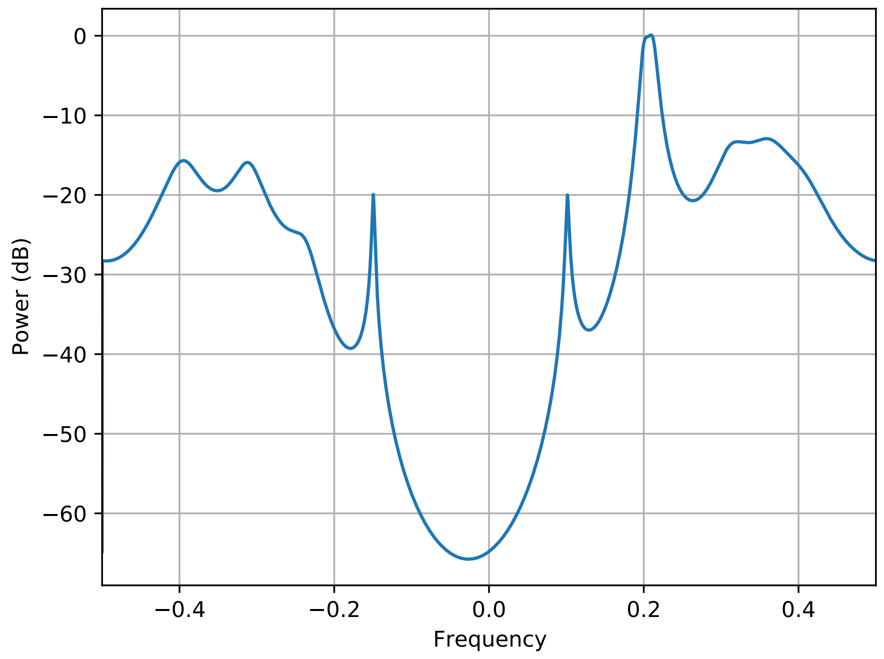
class pminvar(*data*, *order*, *NFFT=None*, *sampling=1.0*, *scale_by_freq=False*)

Class to create PSD based on the Minimum variance spectral estimation

See `minvar()` for description.

```
from spectrum import *
p = pminvar(marple_data, 15, NFFT=4096)
p.plot(sides='centerdc')
```


0
−10
−20
−30
−40
−50
−60
−0.4
−0.2
0.0
0.2
0.4


0
−10
−20
−30
−40
−50
−60
Power (dB)
−0.4
−0.2
0.0
0.2
0.4
Frequency

Constructor

For a detailled description of the parameters, see *minvar()*.

Parameters

- **data** (*array*) – input data (list or numpy.array)
- **order** (*int*) –
- **NFFT** (*int*) – total length of the final data sets (padded with zero if needed; default is 4096)
- **sampling** (*float*) – sampling frequency of the input `data`.

5.4.4 Modified Covariance method

Module related to the modcovar methods

modcovar (*x, order*)

Simple and fast implementation of the covariance AR estimate

This code is 10 times faster than *modcovar_marple()* and more importantly only 10 lines of code, compared to a 200 loc for *modcovar_marple()*

Parameters

- **X** – Array of complex data samples
- **order** (*int*) – Order of linear prediction model

Returns

- P - Real linear prediction variance at order IP
- A - Array of complex linear prediction coefficients

```
from spectrum import modcovar, marple_data, arma2psd, cshift
from pylab import log10, linspace, axis, plot

a, p = modcovar(marple_data, 15)
PSD = arma2psd(a)
PSD = cshift(PSD, len(PSD)/2) # switch positive and negative freq
plot(linspace(-0.5, 0.5, 4096), 10*log10(PSD/max(PSD)))
axis([-0.5,0.5,-60,0])
```

See also:

pmodcovar

Validation the AR parameters are the same as those returned by a completely different function *modcovar_marple()*.

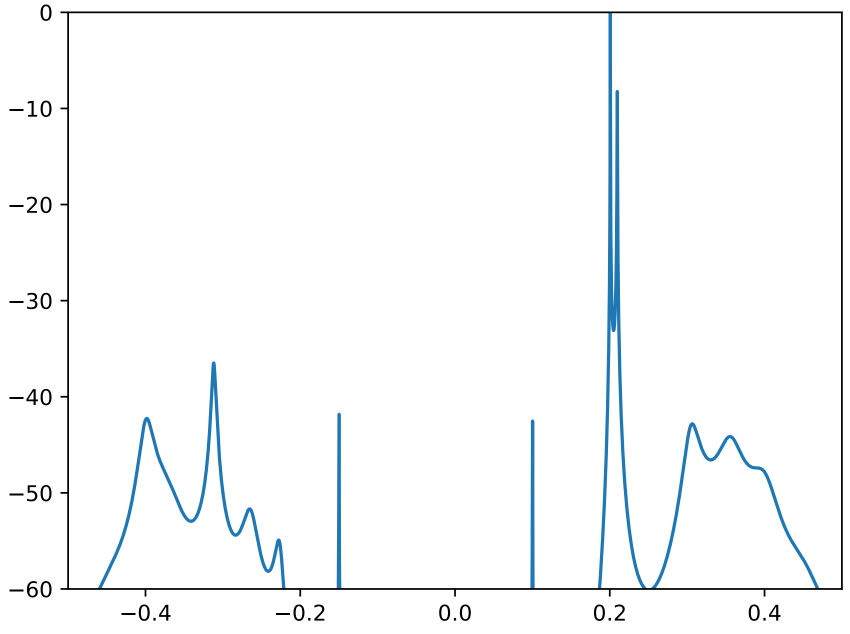
References Mathworks

modcovar_marple (*X, IP*)

Fast algorithm for the solution of the modified covariance least squares normal equations.

This implementation is based on [Marple]. This code is far more complicated and slower than *modcovar()* function, which is now the official version. See *modcovar()* for a detailed description of Modified Covariance method.

Parameters



- **X** –
  - Array of complex data samples X(1) through X(N)
- **IP** (*int*) –
  - Order of linear prediction model (integer)

Returns

- P - Real linear prediction variance at order IP
- A - Array of complex linear prediction coefficients
- **ISTAT - Integer status indicator at time of exit**

  0. for normal exit (no numerical ill-conditioning)
  1. if P is not a positive value
  2. if DELTA' and GAMMA' do not lie in the range 0 to 1
  3. if P' is not a positive value
  4. if DELTA and GAMMA do not lie in the range 0 to 1

Validation the AR parameters are the same as those returned by a completely different function `modcovar()`.

Note: validation. results similar to test example in Marple but starts to differ for ip~8. with ratio of 0.975 for ip=15 probably due to precision.

References [Marple]

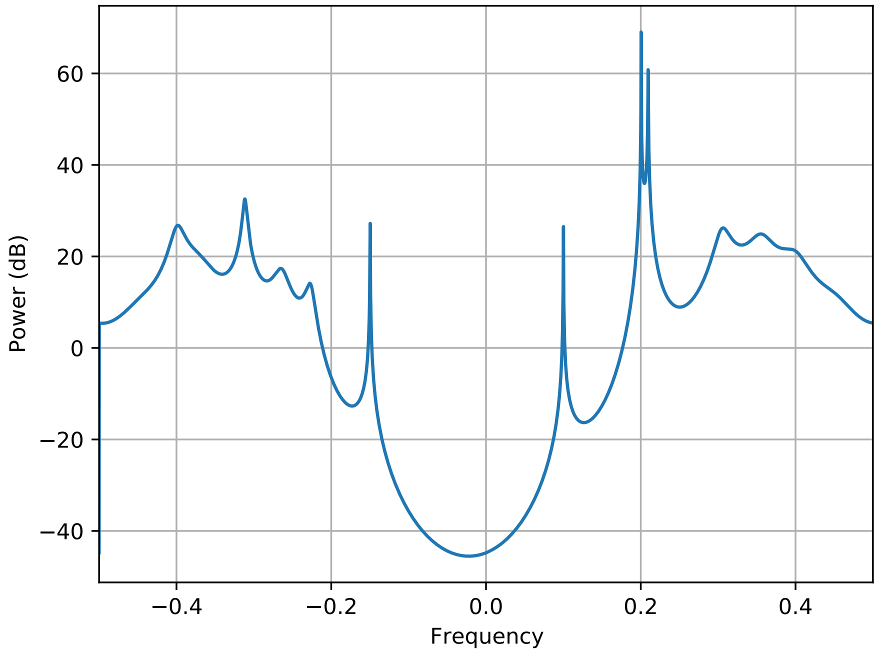
class pmodcovar(*data*, *order*, *NFFT=None*, *sampling=1.0*, *scale_by_freq=False*)

Class to create PSD based on modified covariance algorithm

See *modcovar()* for description.

Examples

```
from spectrum import pmodcovar, marple_data
p = pmodcovar(marple_data, 15, NFFT=4096)
p.plot(sides='centerdc')
```



See also:

modcovar

Constructor

For a detailled description of the parameters, see *modcovar()*.

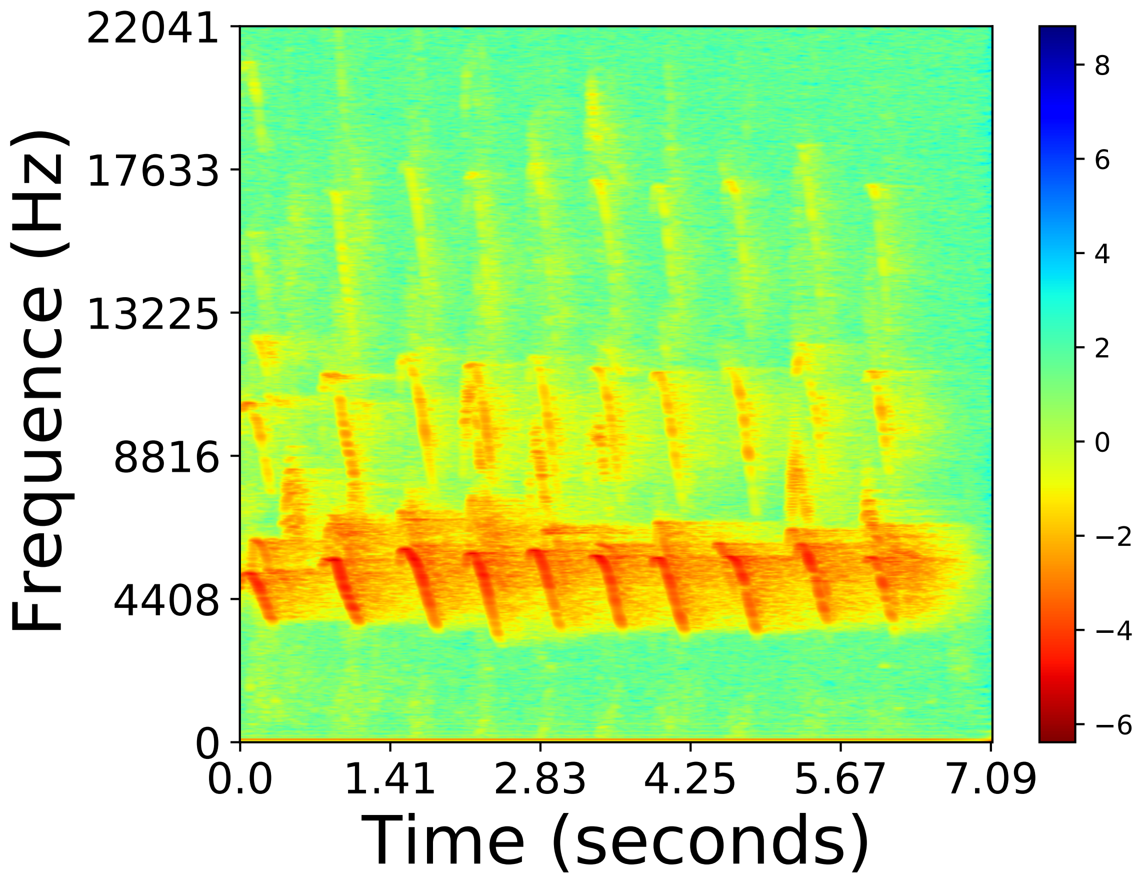
Parameters

- **data** (*array*) – input data (list or numpy.array)
- **order** (*int*) –
- **NFFT** (*int*) – total length of the final data sets (padded with zero if needed; default is 4096)
- **sampling** (*float*) – sampling frequency of the input `data`.

5.4.5 Spectrogram

class Spectrogram(*signal*, *ws=128*, *W=4096*, *sampling=1*, *channel=1*)

Simple example of spectrogram



Warning: this is a prototype and need careful checking about x/y axis

periodogram()

plot(*filename=None*, *vmin=None*, *vmax=None*, *cmap='jet_r'*)

pmtm()

5.5 Tools and classes

Contents

5.5.1 Classes

This module provides the Base class for PSDs

class Spectrum(*data*, *data_y=None*, *sampling=1.0*, *detrend=None*, *scale_by_freq=True*, *NFFT=None*)

Base class for all Spectrum classes

All PSD classes should inherits from this class to store common attributes such as the input data or sampling frequency. An instance is created as follows:

```
>>> p = Spectrum(data, sampling=1024)
>>> p.data
>>> p.sampling
```

The input parameters are:

Parameters

- **data** (*array*) – input data (list or numpy.array)
- **data_y** (*array*) – input data required to perform cross-PSD only.
- **sampling** (`float`) – sampling frequency of the input `data`
- **detrend** (`str`) – detrend method ([None,'mean']) to apply on the input data before computing the PSD. See `detrend`.
- **scale_by_freq** (`bool`) – divide the final PSD by $2 * \pi / df$
- **NFFT** (`int`) – total length of the final data sets (padded with zero if needed; default is 4096)

The input parameters are available as attributes. Additional attributes such as `N` (the data length), `df` (the frequency step are set (see constructor documentation for a complete list).

Warning: `Spectrum` does not compute the PSD estimate.

You can populate manually the `psd` attribute but you should respect the following convention:

- if the input data is real, the PSD is assumed to be one-sided (odd length)
- if the input data is complex, the PSD is assumed to be two-sided (even length).

When `psd` is set, `sides` is reset to its default value, `NFFT` and `df` are updated.

Spectrum instances have plotting utilities like `plot()` that take care of plotting the PSD versus the appropriate frequency range (based on `sampling`, `NFFT` and `sides`)

Note: the modification of some attributes (e.g., NFFT), makes the PSD obsolete. In such cases, the PSD must be re-computed before using `plot()` again.

At any time, you can get general information about the Spectrum instance:

```
>>> p = Spectrum(marple_data)
>>> print(p)
Spectrum summary
    Data length is 64
    PSD not yet computed
    Sampling 1.0
    freq resolution 0.015625
    datatype is complex
    sides is twosided
    scal_by_freq is True
```

Constructor

Attributes:

From the input parameters, the following attributes are set:

- `data` (updates `N`, `df`, `datatype`)
- `data_y` used for cross PSD only (correlogram)
- `detrend`
- `sampling` (updates `df`)
- `scale_by_freq`
- `NFFT` (reset `sides`, `df`)

The following read-only attributes are set during the initialisation:

- `datatype`
- `df`
- `N`

And finally, additional read-write attributes are available:

- `psd`: used to store the PSD data array, which size depends on `sides` i.e., one-sided for real data and two-sided for the complex data.
- `sides`: if set, changed the `psd`.

N

Getter to the original data size. `N` is automatically updated when changing the data only.

NFFT

Getter/Setter to the NFFT attribute.

Parameters **NFFT** – a user choice for setting *NFFT*.

- if None, the NFFT is set to *N*, the data length.
- if 'nextpow2', computes the next power of 2 greater than or equal to the data length.
- if a integer is provided, it must be positive

If NFFT is changed, *sides* is reset and *df* as well.

data

Getter/Setter for the input data. If input is a list, it is cast into a numpy.array. Then, *N*, *df* and *datatype* are updated.

data_y

Getter/Setter to the Y-data

datatype

Getter to the datatype ('real' or 'complex'). *datatype* is automatically updated when changing the data.

detrend

Getter/Setter to detrend:

- None: do not perform any detrend.
- 'mean': remove the mean value of each segment from each segment of the data.
- 'long-mean': remove the mean value from the data before splitting it into segments.
- 'linear': remove linear trend from each segment.

df

Getter to step frequency. This attribute is updated as soon as *data* or *sampling* is changed

frequencies(*sides=None*)

Return the frequency vector according to *sides*

get_converted_psd(*sides*)

This function returns the PSD in the **sides** format

Parameters **sides** (*str*) – the PSD format in ['onesided', 'twosided', 'centerdc']

Returns the expected PSD.

```
from spectrum import *
p = pcovar(marple_data, 15)
centerdc_psd = p.get_converted_psd('centerdc')
```

Note: this function does not change the object, in particular, it does not change the *psd* attribute. If you want to change the psd on the fly, change the attribute *sides*.

method

def __call__(self, *args, **kargs): # To be use with care. THis function is there just to help, it # does not populate the proper attribute except psd. if self.method is not None:

res = self.method(self.data, *args, **kargs) self.psd = res[0]

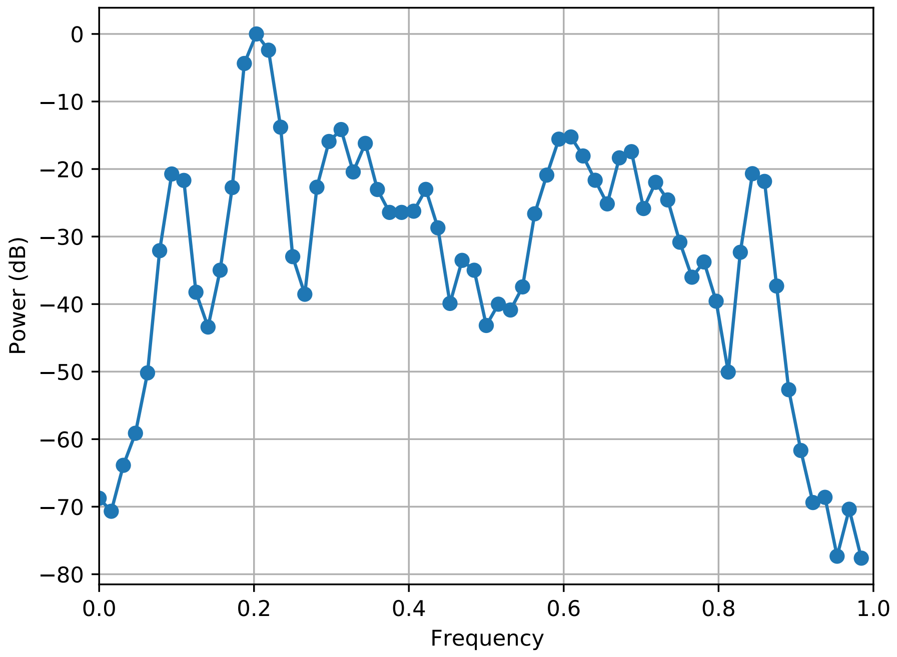
#return res

plot (*filename=None*, *norm=False*, *ylim=None*, *sides=None*, ***kargs*)
a simple plotting routine to plot the PSD versus frequency.

Parameters

- **filename** (*str*) – save the figure into a file
- **norm** – False by default. If True, the PSD is normalised.
- **ylim** – readjust the y range .
- **sides** – if not provided, *sides* is used. See *sides* for details.
- **kargs** – any optional argument accepted by `pylab.plot()`.

```
from spectrum import *
p = Periodogram(marple_data)
p.plot(norm=True, marker='o')
```



power ()
Return the power contained in the PSD

if scale_by_freq is False, the power is:

$$P = N\sum_{k=1}^{N} P_{xx}(k)$$

else, it is

$$P = \sum_{k=1}^{N} P_{xx}(k) \frac{df}{2\pi}$$

Todo: check these equations

psd
Getter/Setter to *psd*

Parameters **psd** (*array*) – the array must be in agreement with the onesided/twosided convention: if the data in real, the psd must be onesided. If the data is complex, the psd must be twosided.

When you set this attribute, several attributes are set:

- *sides* is set to onesided if datatype is real and twosided if datatype is complex.
- *NFFT* is set to len(psd)*2 if sides=onesided and (len(psd)-1)*2 if sides=twosided.
- *range*.N is set to NFFT, which update *df*.

range
Read only attribute to a *Range* object.

run (**args*, ***kargs*)

sampling
Getter/Setter to sampling frequency. Updates the *df* automatically.

scale ()

scale_by_freq
scale the PSD by $2 * \pi / df$

sides
Getter/Setter to the *sides* attributes.

It can be 'onesided', 'twosided', 'centerdc'. This setter changes *psd* to reflect the user argument choice.

If the datatype is complex, sides cannot be one-sided.

class FourierSpectrum (*data*, *sampling=1.0*, *window='hanning'*, *NFFT=None*, *detrend=None*, *scale_by_freq=True*, *lag=-1*)
Spectrum based on Fourier transform.

This class inherits attributes and methods from *Spectrum*. It is used by children class *Periodogram*, *pcorrelogram* and `Welch` PSD estimates.

The parameters are those used by *Spectrum*

Parameters

- **data** (*array*) – Input data (list or numpy.array)
- **data_y** – input data required to perform cross-PSD only.
- **sampling** (*float*) – sampling frequency of the input `data`
- **detrend** (*str*) – detrend method ([None,'mean']) to apply on the input data before computing the PSD. See `detrend`.
- **scale_by_freq** (*bool*) – Divide the final PSD by $2 * \pi / df$
- **NFFT** (*int*) – total length of the data given to the FFT

In addition you need specific parameters such as:

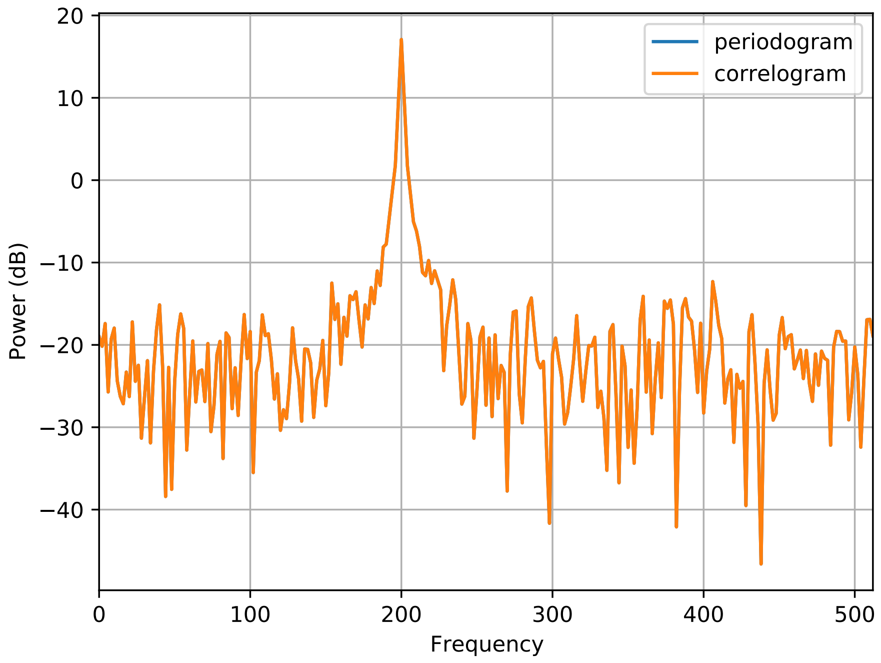
Parameters

- **window** (*str*) – a tapering window. See *Window*.

- **lag** (*int*) – to be used by the *pcorrelogram* methods only.

This class has dedicated PSDs methods such as `speriodogram()`, which are equivalent to children class such as *Periodogram*.

```
from spectrum import datasets
from spectrum import FourierSpectrum
s = FourierSpectrum(datasets.data_cosine(), lag=32, sampling=1024, NFFT=512)
s.periodogram()
s.plot(label='periodogram')
#s.correlogram()
s.plot(label='correlogram')
from pylab import legend, xlim
legend()
xlim(0,512)
```



Constructor

See the class documentation for the parameters.

Additional attributes to those inherited from

Spectrum are:

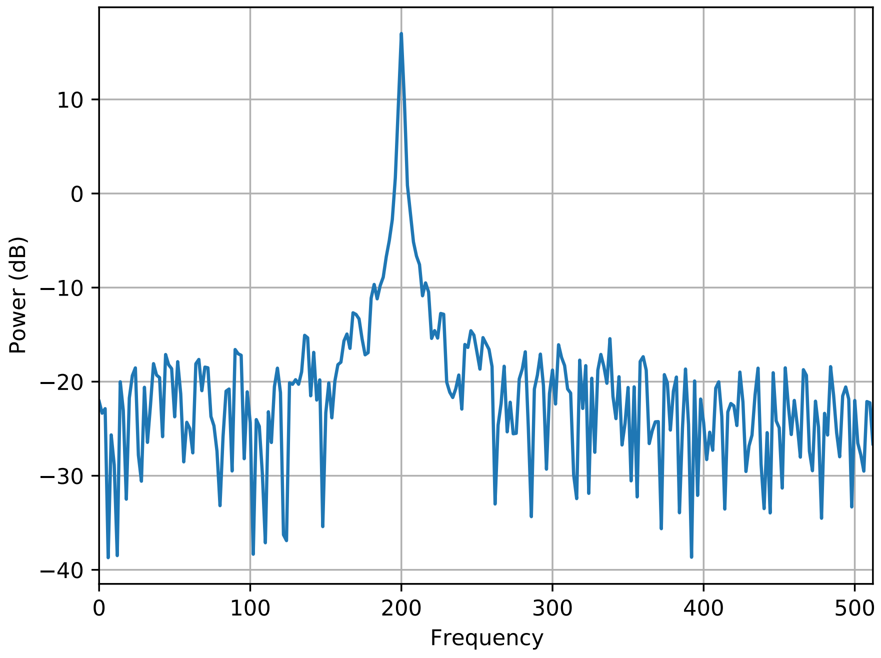
- *lag*, a lag used to compute the autocorrelation
- *window*, the tapering window to be used

lag
    Getter/Setter used by the correlogram when autocorrelation estimates are required.

periodogram()
    An alias to *Periodogram*

The parameters are extracted from the attributes. Relevant attributes ares `window`, attr:*sampling*, attr:*NFFT*, attr:*scale_by_freq*, `detrend`.

```
from spectrum import datasets
from spectrum import FourierSpectrum
s = FourierSpectrum(datasets.data_cosine(), sampling=1024, NFFT=512)
s.periodogram()
s.plot()
```



window
Tapering window to be applied

class ParametricSpectrum(*data*, *sampling=1.0*, *ar_order=None*, *ma_order=None*, *lag=-1*, *NFFT=None*, *detrend=None*, *scale_by_freq=True*)
Spectrum based on Fourier transform.

This class inherits attributes and methods from `Spectrum`. It is used by children class `Periodogram`, `pcorrelogram` and `Welch` PSD estimates. The parameters are those used by `Spectrum`.

Parameters

- **data** (*array*) – Input data (list or numpy.array)
- **sampling** (*float*) – sampling frequency of the input `data`
- **detrend** (*str*) – detrend method ([None,'mean']) to apply on the input data before computing the PSD. See `detrend`.
- **scale_by_freq** (*bool*) – Divide the final PSD by $2 * \pi / df$

In addition you need specific parameters such as:

Parameters

- **window** (*str*) – a tapering window. See `Window`.

- **lag** (*int*) – to be used by the `pcorrelogram` methods only.
- **NFFT** (*int*) – Total length of the data given to the FFT

This class has dedicated PSDs methods such as `periodogram()`, which are equivalent to children class such as `Periodogram`.

```
from spectrum import datasets
from spectrum import ParametricSpectrum
data = datasets.data_cosine(N=1024)
s = ParametricSpectrum(data, ar_order=4, ma_order=4, sampling=1024, NFFT=512,␣
↪lag=10)
#s.parma()
#s.plot(sides='onesided')
#s.plot(sides='twosided')
```

Constructor

See the class documentation for the parameters.

Additional attributes to those inherited from `Spectrum`:

- `ar_order`, the ar order of the PSD estimates
- `ma_order`, the ar order of the PSD estimates

ar

ar_order

ma

ma_order

plot_reflection()

reflection

self.reflection = None elif method == 'aryule':

from spectrum import aryule ar, v, coeff = aryule(self.data, self.ar_order) self.ar = ar self.rho = v self.reflection = coeff

rho

class Range(*N*, *sampling=1.0*)

A class to ease the creation of frequency ranges.

Given the length `N` of a data sample and a sampling frequency `sampling`, this class provides methods to generate frequency ranges

- `centerdc()`: frequency range from -sampling/2 up to sampling/2 (excluded),
- `twosided()`: frequency range from 0 up to sampling (excluded),
- `onesided()`: frequency range from 0 up to sampling (included or excluded depending on evenness of the data). If NFFT is even, PSD has length NFFT/2 + 1 over the interval [0,pi]. If NFFT is odd, the length of PSD is (NFFT+1)/2 and the interval is [0, pi)

Each method has a generator version:

```
>>> r = Range(10, sampling=1)
>>> list(r.onesided_gen())
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5]
>>> r.onesided()
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5]
```

The frequency range length is $N/2+1$ for the *onesided* case ($(N+1)/2$ if N is odd), and N for the *twosided* and *centerdc* cases:

```
>>> r.twosided()
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
>>> len(r.twosided())
10
>>> len(r.centerdc())
10
>>> len(r.onesided())
5
```

Constructor

Parameters

- **N** (*int*) – the data length
- **sampling** (*float*) – sampling frequency of the input `data`.

Attributes:

From the input parameters, read/write attributes are set:

- *N*, the data length,
- *sampling*, the sampling frequency.

Additionally, the following read-only attribute is available:

- *df*, the frequency step computed from *N* and *sampling*.

N

Getter/Setter of the data length. If changed, *df* is updated.

centerdc()

Return the two-sided frequency range as a list (see *centerdc_gen()* for details).

centerdc_gen()

Return the centered frequency range as a generator.

```
>>> print(list(Range(8).centerdc_gen()))
[-0.5, -0.375, -0.25, -0.125, 0.0, 0.125, 0.25, 0.375]
```

df

Getter to access the frequency step, computed from *N* and *sampling*.

onesided()

Return the one-sided frequency range as a list (see *onesided_gen()* for details).

onesided_gen()

Return the one-sided frequency range as a generator.

If `N` is even, the length is N/2 + 1. If `N` is odd, the length is (N+1)/2.

```
>>> print(list(Range(8).onesided()))
[0.0, 0.125, 0.25, 0.375, 0.5]
>>> print(list(Range(9).onesided()))
[0.0, 0.1111, 0.2222, 0.3333, 0.4444]
```

sampling
Getter/Setter of the sampling frequency. If changed, `df` is updated.

twosided()
Return the two-sided frequency range as a list (see `twosided_gen()` for details).

twosided_gen()
Returns the twosided frequency range as a generator

```
>>> print(list(Range(8).centerdc_gen()))
[0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875]
```

5.5.2 Correlation

Correlation module

Provides two correlation functions. `CORRELATION()` is slower than `xcorr()`. However, the output is as expected by some other functions. Ultimately, it should be replaced by `xcorr()`.

For real data, the behaviour of the 2 functions is identical. However, for complex data, xcorr returns a 2-sides correlation.

`CORRELATION`(x[, y, maxlags, norm])	Correlation function
`xcorr`(x[, y, maxlags, norm])	Cross-correlation using numpy.correlate

CORRELATION(*x*, *y=None*, *maxlags=None*, *norm='unbiased'*)
Correlation function

This function should give the same results as `xcorr()` but it returns the positive lags only. Moreover the algorithm does not use FFT as compared to other algorithms.

Parameters

- **x** (*array*) – first data array of length N
- **y** (*array*) – second data array of length N. If not specified, computes the autocorrelation.
- **maxlags** (*int*) – compute cross correlation between [0:maxlags] when maxlags is not specified, the range of lags is [0:maxlags].
- **norm** (*str*) – normalisation in ['biased', 'unbiased', None, 'coeff']
  - *biased* correlation=raw/N,
  - *unbiased* correlation=raw/(N-|*lag*|)
  - *coeff* correlation=raw/(rms(x).rms(y))/N
  - None correlation=raw

Returns

- a numpy.array correlation sequence, r[1,N]
- a float for the zero-lag correlation, r[0]

The *unbiased* correlation has the form:

$$\hat{r}_{xx} = \frac{1}{N-m}T \sum_{n=0}^{N-m-1} x[n+m]x^*[n]T$$

The *biased* correlation differs by the front factor only:

$$\check{r}_{xx} = \frac{1}{N}T \sum_{n=0}^{N-m-1} x[n+m]x^*[n]T$$

with $0 \leq m \leq N-1$.

```
>>> from spectrum import CORRELATION
>>> x = [1,2,3,4,5]
>>> res = CORRELATION(x,x, maxlags=0, norm='biased')
>>> res[0]
11.0
```

Note: this function should be replaced by `xcorr()`.

See also:

`xcorr()`

`xcorr` (*x*, *y=None*, *maxlags=None*, *norm='biased'*)

Cross-correlation using numpy.correlate

Estimates the cross-correlation (and autocorrelation) sequence of a random process of length N. By default, there is no normalisation and the output sequence of the cross-correlation has a length 2*N+1.

Parameters

- **x** (*array*) – first data array of length N
- **y** (*array*) – second data array of length N. If not specified, computes the autocorrelation.
- **maxlags** (*int*) – compute cross correlation between [-maxlags:maxlags] when maxlags is not specified, the range of lags is [-N+1:N-1].
- **option** (*str*) – normalisation in ['biased', 'unbiased', None, 'coeff']

The true cross-correlation sequence is

$$r_{xy}[m] = E(x[n+m].y^*[n]) = E(x[n].y^*[n-m])$$

However, in practice, only a finite segment of one realization of the infinite-length random process is available.

The correlation is estimated using numpy.correlate(x,y,'full'). Normalisation is handled by this function using the following cases:

- 'biased': Biased estimate of the cross-correlation function
- 'unbiased': Unbiased estimate of the cross-correlation function
- **'coeff': Normalizes the sequence so the autocorrelations at zero** lag is 1.0.

Returns

- a numpy.array containing the cross-correlation sequence (length 2*N-1)
- lags vector

Note: If x and y are not the same length, the shorter vector is zero-padded to the length of the longer vector.

Examples

```
>>> from spectrum import xcorr
>>> x = [1,2,3,4,5]
>>> c, l = xcorr(x,x, maxlags=0, norm='biased')
>>> c
array([ 11.])
```

See also:

CORRELATION().

5.5.3 Tools

Tools module

db2mag(xdb)	Convert decibels (dB) to magnitude
db2pow(xdb)	Convert decibels (dB) to power
mag2db(x)	Convert magnitude to decibels (dB)
nextpow2(x)	returns the smallest power of two that is greater than or equal to the absolute value of x.
pow2db(x)	returns the corresponding decibel (dB) value for a power value x.
onesided_2_twosided(data)	Convert a two-sided PSD to a one-sided PSD
twosided_2_onesided(data)	Convert a one-sided PSD to a twosided PSD
centerdc_2_twosided(data)	Convert a center-dc PSD to a twosided PSD
twosided_2_centerdc(data)	Convert a two-sided PSD to a center-dc PSD

Code author: Thomas Cokelaer, 2011

centerdc_2_twosided(*data*)
Convert a center-dc PSD to a twosided PSD

cshift(*data, offset*)
Circular shift to the right (within an array) by a given offset

Parameters

- **data** (*array*) – input data (list or numpy.array)
- **offset** (*int*) – shift the array with the offset

```
>>> from spectrum import cshift
>>> cshift([0, 1, 2, 3, -2, -1], 2)
array([-2, -1,  0,  1,  2,  3])
```

db2mag (*xdb*)

Convert decibels (dB) to magnitude

```
>>> from spectrum import db2mag
>>> db2mag(-20)
0.1
```

See also:

pow2db()

db2pow (*xdb*)

Convert decibels (dB) to power

```
>>> from spectrum import db2pow
>>> p = db2pow(-10)
>>> p
0.1
```

See also:

pow2db()

fftshift (*x*)

wrapper to numpy.fft.fftshift

```
>>> from spectrum import fftshift
>>> x = [100, 2, 3, 4, 5]
>>> fftshift(x)
array([  4,   5, 100,   2,   3])
```

log10 (*data*)

mag2db (*x*)

Convert magnitude to decibels (dB)

The relationship between magnitude and decibels is:

$$X_{dB} = 20 * \log_{10}(x)$$

```
>>> from spectrum import mag2db
>>> mag2db(0.1)
-20.0
```

See also:

db2mag()

nextpow2 (*x*)

returns the smallest power of two that is greater than or equal to the absolute value of x.

This function is useful for optimizing FFT operations, which are most efficient when sequence length is an exact power of two.

Example

```
>>> from spectrum import nextpow2
>>> x = [255, 256, 257]
>>> nextpow2(x)
array([8, 8, 9])
```

onesided_2_twosided(*data*)

Convert a two-sided PSD to a one-sided PSD

In order to keep the power in the twosided PSD the same as in the onesided version, the twosided values are 2 times lower than the input data (except for the zero-lag and N-lag values).

```
>>> twosided_2_onesided([10, 4, 6, 8])
array([ 10.,   2.,   3.,   3., 2., 8.])
```

pow2db(*x*)

returns the corresponding decibel (dB) value for a power value x.

The relationship between power and decibels is:

$$X_{dB} = 10 * \log_{10}(x)$$

```
>>> from spectrum import pow2db
>>> x = pow2db(0.1)
>>> x
-10.0
```

twosided(*data*)

return a twosided vector with non-duplication of the first element

```
>>> from spectrum import twosided
>>> a = [1,2,3]
>>> twosided(a)
array([3, 2, 1, 2, 3])
```

twosided_2_centerdc(*data*)

Convert a two-sided PSD to a center-dc PSD

twosided_2_onesided(*data*)

Convert a one-sided PSD to a twosided PSD

In order to keep the power in the onesided PSD the same as in the twosided version, the onesided values are twice as much as in the input data (except for the zero-lag value).

```
>>> twosided_2_onesided([10, 2,3,3,2,8])
array([ 10.,   4.,   6.,   8.])
```

5.5.4 datasets

The datasets module provides data sets to test the Spectrum functionalities.

data_cosine([N, A, sampling, freq])	Return a noisy cosine at a given frequency.
marple_data	64-complex data length from Marple reference [Marple]

TimeSeries(data[, sampling])	A simple Base Class for various data sets.

Code author: Thomas Cokelaer 2011

Reference [Marple]

class TimeSeries(*data*, *sampling=1*)
A simple Base Class for various data sets.

```
>>> from spectrum import TimeSeries
>>> data = [1, 2, 3, 4, 3, 2, 1, 0 ]
>>> ts = TimeSeries(data, sampling=1)
>>> ts.plot()
>>> ts.dt
1.0
```

Parameters

- **data** (*array*) – input data (list or numpy.array)
- **sampling** – the sampling frequency of the data (default 1Hz)

plot(***kargs*)
Plot the data set, using the sampling information to set the x-axis correctly.

data_cosine(*N=1024*, *A=0.1*, *sampling=1024.0*, *freq=200*)
Return a noisy cosine at a given frequency.

Parameters

- **N** – the final data size
- **A** – the strength of the noise
- **sampling** (*float*) – sampling frequency of the input `data`.
- **freq** (*float*) – the frequency f_0 of the cosine.

$$x[t] = cos(2\pi t * f_0) + Aw[t]$$

where w[t] is a white noise of variance 1.

```
>>> from spectrum import data_cosine
>>> a = data_cosine(N=1024, sampling=1024, A=0.5, freq=100)
```

data_two_freqs(*N=200*)
A simple test example with two close frequencies

dolphin_filename = '/home/docs/checkouts/readthedocs.org/user_builds/pyspectrum/envs/latest
filename of a WAV data file 150,000 data points

marple_data = [(1.349839091+2.011167288j), (-2.117270231+0.817693591j), (-1.786421657-1.29
64-complex data length from Marple reference [Marple]

randn(*d0*, *d1*, *...*, *dn*)
Return a sample (or samples) from the "standard normal" distribution.

Note: This is a convenience function for users porting code from Matlab, and wraps *standard_normal*. That function takes a tuple to specify the size of the output, which is consistent with other NumPy functions like *numpy.zeros* and *numpy.ones*.

Note: New code should use the `standard_normal` method of a `default_rng()` instance instead; please see the random-quick-start.

If positive int_like arguments are provided, *randn* generates an array of shape `(d0, d1, ..., dn)`, filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1. A single float randomly sampled from the distribution is returned if no argument is provided.

Parameters

d0, d1, . . . , dn [int, optional] The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

Returns

Z [ndarray or float] A `(d0, d1, ..., dn)`-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

See also:

`standard_normal` Similar, but takes a tuple as its argument.

`normal` Also accepts mu and sigma arguments.

`Generator.standard_normal` which should be used for new code.

Notes

For random samples from $N(\mu, \sigma^2)$, use:

`sigma * np.random.randn(...) + mu`

Examples

```
>>> np.random.randn()
2.1923875335537315  # random
```

Two-by-four array of samples from N(3, 6.25):

```
>>> 3 + 2.5 * np.random.randn(2, 4)
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```

`spectrum_data`(*filename*)
Simple utilities to retrieve data sets from

5.5.5 Linear Algebra Tools

cholesky

Cholesky methods

CHOLESKY(A, B[, method])	Solve linear system *AX=B* using CHOLESKY method.

Code author: Thomas Cokelaer, 2011

CHOLESKY (*A*, *B*, *method='scipy'*)
Solve linear system *AX=B* using CHOLESKY method.

Parameters

- **A** – an input Hermitian matrix
- **B** – an array
- **method** (*str*) – a choice of method in [numpy, scipy, numpy_solver]
  - *numpy_solver* relies entirely on numpy.solver (no cholesky decomposition)
  - *numpy* relies on the numpy.linalg.cholesky for the decomposition and numpy.linalg.solve for the inversion.
  - *scipy* uses scipy.linalg.cholesky for the decomposition and scipy.linalg.cho_solve for the inversion.

Description

When a matrix is square and Hermitian (symmetric with lower part being the complex conjugate of the upper one), then the usual triangular factorization takes on the special form:

$$A = RR^H$$

where R is a lower triangular matrix with nonzero real principal diagonal element. The input matrix can be made of complex data. Then, the inversion to find x is made as follows:

$$Ry = B$$

and

$$Rx = y$$

```
>>> import numpy
>>> from spectrum import CHOLESKY
>>> A = numpy.array([[ 2.0+0.j ,  0.5-0.5j, -0.2+0.1j],
...     [ 0.5+0.5j,  1.0+0.j ,  0.3-0.2j],
...     [-0.2-0.1j,  0.3+0.2j,  0.5+0.j ]])
>>> B = numpy.array([ 1.0+3.j ,  2.0-1.j ,  0.5+0.8j])
>>> CHOLESKY(A, B)
array([ 0.95945946+5.25675676j,  4.41891892-7.04054054j,
       -5.13513514+6.35135135j])
```

eigen

MINEIGVAL(*T0*, *T*, *TOL*)
Finds the minimum eigenvalue of a Hermitian Toeplitz matrix

The classical power method is used together with a fast Toeplitz equation solution routine. The eigenvector is normalized to unit length.

Parameters

- **T0** – Scalar corresponding to real matrix element t(0)
- **T** – Array of M complex matrix elements t(1),…,t(M) C from the left column of the Toeplitz matrix
- **TOL** – Real scalar tolerance; routine exits when [EVAL(k) - EVAL(k-1)]/EVAL(k-1) < TOL , where the index k denotes the iteration number.

Returns

- EVAL - Real scalar denoting the minimum eigenvalue of matrix
- EVEC - Array of M complex eigenvector elements associated

Note:

- External array T must be dimensioned >= M
- array EVEC must be >= M+1
- Internal array E must be dimensioned >= M+1 .
- **dependencies**
  - *spectrum.toeplitz.HERMTOEP()*

levinson

Levinson module

LEVINSON(r[, order, allow_singularity])	Levinson-Durbin recursion.

Code author: Thomas Cokelaer, 2011

LEVINSON(*r*, *order=None*, *allow_singularity=False*)
Levinson-Durbin recursion.

Find the coefficients of a length(r)-1 order autoregressive linear process

Parameters

- **r** – autocorrelation sequence of length N + 1 (first element being the zero-lag autocorrelation)
- **order** – requested order of the autoregressive coefficients. default is N.
- **allow_singularity** – false by default. Other implementations may be True (e.g., octave)

Returns

- the *N+1* autoregressive coefficients $A = (1, a_1...a_N)$
- the prediction errors
- the *N* reflections coefficients values

This algorithm solves the set of complex linear simultaneous equations using Levinson algorithm.

$$\mathbf{T}_M \left(\begin{array}{c} 1 \\ \mathbf{a}_M \end{array} \right) = \left(\begin{array}{c} \rho_M \\ \mathbf{0}_M \end{array} \right)$$

where $\mathbf{T}_M$ is a Hermitian Toeplitz matrix with elements $T_0, T_1, \dots, T_M$.

Note: Solving this equations by Gaussian elimination would require M^3 operations whereas the levinson algorithm requires $M^2 + M$ additions and $M^2 + M$ multiplications.

This is equivalent to solve the following symmetric Toeplitz system of linear equations

$$\left(\begin{array}{cccc} r_1 & r_2^* & \dots & r_n^* \\ r_2 & r_1^* & \dots & r_{n-1}^* \\ \dots & \dots & \dots & \dots \\ r_n & \dots & r_2 & r_1 \end{array} \right) \left(\begin{array}{c} a_2 \\ a_3 \\ \dots \\ a_{N+1} \end{array} \right) = \left(\begin{array}{c} -r_2 \\ -r_3 \\ \dots \\ -r_{N+1} \end{array} \right)$$

where $r = (r_1...r_{N+1})$ is the input autocorrelation vector, and r_i^* denotes the complex conjugate of r_i. The input r is typically a vector of autocorrelation coefficients where lag 0 is the first element r_1.

```
>>> import numpy; from spectrum import LEVINSON
>>> T = numpy.array([3., -2+0.5j, .7-1j])
>>> a, e, k = LEVINSON(T)
```

rlevinson(*a*, *efinal*)

computes the autocorrelation coefficients, R based on the prediction polynomial A and the final prediction error Efinal, using the stepdown algorithm.

Works for real or complex data

Parameters

- **a** –
- **efinal** –

Returns

- R, the autocorrelation
- U prediction coefficient
- kr reflection coefficients
- e errors

A should be a minimum phase polynomial and A(1) is assumed to be unity.

Returns

(P+1) by (P+1) upper triangular matrix, U, that holds the i'th order prediction polynomials Ai, i=1:P, where P is the order of the input polynomial, A.

[1 a1(1)* a2(2)* ….. aP(P) *] [0 1 a2(1)* ….. aP(P-1)*]

U = [................................] [0 0 0 ….. 1] from which the i'th order prediction polynomial can be extracted using Ai=U(i+1:-1:1,i+1)'. The first row of U contains the conjugates of the reflection coefficients, and the K's may be extracted using, K=conj(U(1,2:end)).

Todo: remove the conjugate when data is real data, clean up the code test and doc.

toeplitz

Toeplitz module

These functions are not yet used by other functions, which explains the lack of documentation, test, examples.

Nevertheless, they can be used for production.

HERMTOEP(T0, T, Z)	solve Tx=Z by a variation of Levinson algorithm where T is a complex hermitian toeplitz matrix

Code author: Thomas Cokelaer, 2011

HERMTOEP (*T0*, *T*, *Z*)
solve Tx=Z by a variation of Levinson algorithm where T is a complex hermitian toeplitz matrix

Parameters

- **T0** – zero lag value
- **T** – r1 to rN

Returns X

used by eigen PSD method

linalg

Linear algebra tools

csvd(A)	SVD decomposition using numpy.linalg.svd
corrmtx(x_input, m[, method])	Correlation matrix
pascal(n)	Return Pascal matrix

Code author: Thomas Cokelaer 2011

pascal (*n*)
Return Pascal matrix

Parameters **n** (*int*) – size of the matrix

```
>>> from spectrum import pascal
>>> pascal(6)
array([[   1.,    1.,    1.,    1.,    1.,    1.],
       [   1.,    2.,    3.,    4.,    5.,    6.],
```

(continues on next page)

(continued from previous page)

```
        [   1.,    3.,    6.,   10.,   15.,   21.],
        [   1.,    4.,   10.,   20.,   35.,   56.],
        [   1.,    5.,   15.,   35.,   70.,  126.],
        [   1.,    6.,   21.,   56.,  126.,  252.]])
```

Todo: use the symmetric property to improve computational time if needed

csvd(*A*)

SVD decomposition using numpy.linalg.svd

Parameters **A** – a M by N matrix

Returns

- U, a M by M matrix
- S the N eigen values
- V a N by N matrix

See `numpy.linalg.svd()` for a detailed documentation.

Should return the same as in [Marple] , CSVD routine.

```
U, S, V = numpy.linalg.svd(A)
U, S, V = cvsd(A)
```

corrmtx(*x_input*, *m*, *method='autocorrelation'*)

Correlation matrix

This function is used by PSD estimator functions. It generates the correlation matrix from a correlation data set and a maximum lag.

Parameters

- **x** (*array*) – autocorrelation samples (1D)
- **m** (*int*) – the maximum lag

Depending on the choice of the method, the correlation matrix has different sizes, but the number of rows is always m+1.

Method can be :

- 'autocorrelation': (default) X is the (n+m)-by-(m+1) rectangular Toeplitz matrix derived using prewindowed and postwindowed data.
- 'prewindowed': X is the n-by-(m+1) rectangular Toeplitz matrix derived using prewindowed data only.
- 'postwindowed': X is the n-by-(m+1) rectangular Toeplitz matrix that derived using postwindowed data only.
- 'covariance': X is the (n-m)-by-(m+1) rectangular Toeplitz matrix derived using nonwindowed data.
- 'modified': X is the 2(n-m)-by-(m+1) modified rectangular Toeplitz matrix that generates an autocorrelation estimate for the length n data vector x, derived using forward and backward prediction error estimates.

Returns

- the autocorrelation matrix

- R, the (m+1)-by-(m+1) autocorrelation matrix estimate `R= X'*X`.

Algorithm details:

The **autocorrelation** matrix is a $(N+p)\times(p+1)$ rectangular Toeplitz data matrix:

$$X_p = \begin{pmatrix} L_p \\ T_p \\ Up \end{pmatrix}$$

where the lower triangular $p\times(p+1)$ matrix L_p is

$$L_p = \begin{pmatrix} x[1] & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ x[p] & \cdots & x[1] & 0 \end{pmatrix}$$

where the rectangular $(N-p)\times(p+1)$ matrix T_p is

$$T_p = \begin{pmatrix} x[p+1] & \cdots & x[1] \\ x[p+2] & \cdots & x[2] \\ \vdots & \ddots & \vdots \\ x[N-1] & \cdots & x[N-p-1] \\ x[N] & \cdots & x[N-p] \end{pmatrix}$$

and where the upper triangular $p\times(p+1)$ matrix U_p is

$$U_p = \begin{pmatrix} 0 & x[N] & \cdots & x[N-p+1] \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & x[N] \end{pmatrix}$$

From this definition, the prewindowed matrix is

$$X_p = \begin{pmatrix} L_p \\ T_p \end{pmatrix}$$

the postwindowed matrix is

$$X_p = \begin{pmatrix} T_p \\ U_p \end{pmatrix}$$

the covariance matrix is:

$$X_p = \begin{pmatrix} T_p \end{pmatrix}$$

and the modified covariance matrix is:

$$X_p = \begin{pmatrix} T_p \\ T_p^* \end{pmatrix}$$

Transfer function

Linear systems

tf2zp(*b*, *a*)

Convert transfer function filter parameters to zero-pole-gain form

Find the zeros, poles, and gains of this continuous-time system:

Warning: b and a must have the same length.

```
from spectrum import tf2zp
b = [2,3,0]
a = [1, 0.4, 1]
[z,p,k] = tf2zp(b,a)          % Obtain zero-pole-gain form
z =
    1.5
    0
p =
   -0.2000 + 0.9798i
    -0.2000 - 0.9798i
k =
   2
```

Parameters

- **b** – numerator
- **a** – denominator
- **fill** – If True, check that the length of a and b are the same. If not, create a copy of the shortest element and append zeros to it.

Returns z (zeros), p (poles), g (gain)

Convert transfer function f(x)=sum(b*x^n)/sum(a*x^n) to zero-pole-gain form f(x)=g*prod(1-z*x)/prod(1-p*x)

Todo: See if tf2ss followed by ss2zp gives better results. These are available from the control system toolbox. Note that the control systems toolbox doesn't bother, but instead uses

See also:

scipy.signal.tf2zpk, which gives the same results but uses a different algorithm (z^-1 instead of z).

eqtflength(*b*, *a*)

Given two list or arrays, pad with zeros the shortest array

Parameters

- **b** – list or array
- **a** – list or array

```
>>> from spectrum.transfer import eqtflength
>>> a = [1,2]
>>> b = [1,2,3,4]
>>> a, b, = eqtflength(a,b)
```

latc2tf()

latcfilt()

ss2zpk(*a, b, c, d, input=0*)
State-space representation to zero-pole-gain representation.

Parameters

- **A** – ndarray State-space representation of linear system.
- **B** – ndarray State-space representation of linear system.
- **C** – ndarray State-space representation of linear system.
- **D** – ndarray State-space representation of linear system.
- **input** (*int*) – optional For multiple-input systems, the input to use.

Returns

- z, p : sequence Zeros and poles.
- k : float System gain.

Note: wrapper of scipy function ss2zpk

tf2sos()

tf2ss()

tf2zpk(*b, a*)
Return zero, pole, gain (z,p,k) representation from a numerator, denominator representation of a linear filter.

Convert zero-pole-gain filter parameters to transfer function form

Parameters

- **b** (*ndarray*) – numerator polynomial.
- **a** (*ndarray*) – numerator and denominator polynomials.

Returns

- z : ndarray Zeros of the transfer function.
- p : ndarray Poles of the transfer function.
- k : float System gain.

If some values of b are too close to 0, they are removed. In that case, a BadCoefficients warning is emitted.

```
>>> import scipy.signal
>>> from spectrum.transfer import tf2zpk
>>> [b, a] = scipy.signal.butter(3.,.4)
>>> z, p ,k = tf2zpk(b,a)
```

See also:

zpk2tf()

Note: wrapper of scipy function tf2zpk

zpk2ss(*z, p, k*)
Zero-pole-gain representation to state-space representation

Parameters

- **z,p** (*sequence*) – Zeros and poles.
- **k** (*float*) – System gain.

Returns

- A, B, C, D : ndarray State-space matrices.

Note: wrapper of scipy function zpk2ss

zpk2tf (*z*, *p*, *k*)

Return polynomial transfer function representation from zeros and poles

Parameters

- **z** (*ndarray*) – Zeros of the transfer function.
- **p** (*ndarray*) – Poles of the transfer function.
- **k** (*float*) – System gain.

Returns b : ndarray Numerator polynomial. a : ndarray Numerator and denominator polynomials.

zpk2tf() forms transfer function polynomials from the zeros, poles, and gains of a system in factored form.

zpk2tf(z,p,k) finds a rational transfer function

$$\frac{B(s)}{A(s)} = \frac{b_1 s^{n-1} + \dots b_{n-1}s + b_n}{a_1 s^{m-1} + \dots a_{m-1}s + a_m}$$

given a system in factored transfer function form

$$H(s) = \frac{Z(s)}{P(s)} = k\frac{(s - z_1)(s - z_2)\dots(s - z_m)}{(s - p_1)(s - p_2)\dots(s - p_n)}$$

with p being the pole locations, and z the zero locations, with as many. The gains for each numerator transfer function are in vector k. The zeros and poles must be real or come in complex conjugate pairs. The polynomial denominator coefficients are returned in row vector a and the polynomial numerator coefficients are returned in matrix b, which has as many rows as there are columns of z.

Inf values can be used as place holders in z if some columns have fewer zeros than others.

Note: wrapper of scipy function zpk2tf

5.5.6 Waveforms

morlet (*lb*, *ub*, *n*)

Generate the Morlet waveform

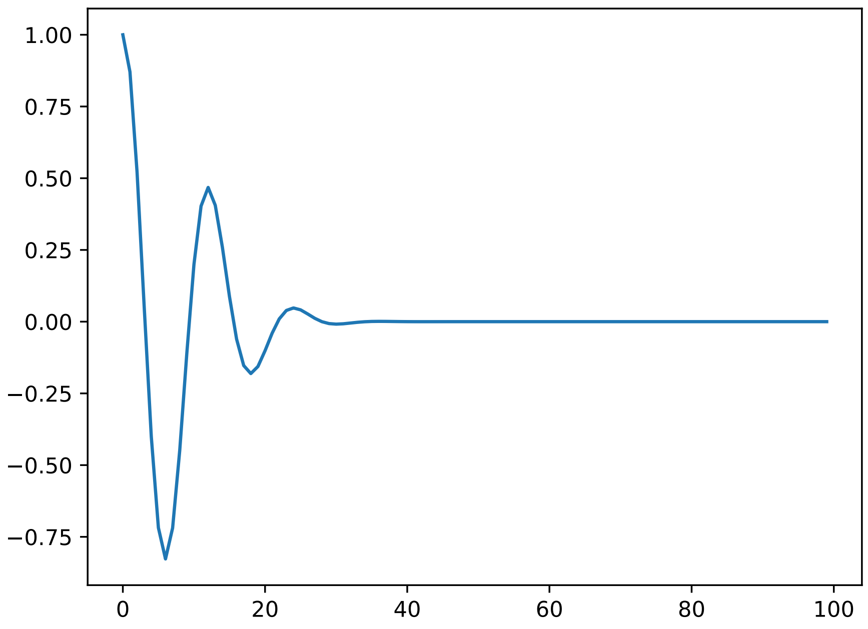
The Morlet waveform is defined as follows:

$$w[x] = \cos 5x \exp^{-x^2/2}$$

Parameters

- **lb** – lower bound
- **ub** – upper bound
- **n** (*int*) – waveform data samples

```
from spectrum import morlet
from pylab import plot
plot(morlet(0,10,100))
```



chirp(*t, f0=0.0, t1=1.0, f1=100.0, form='linear', phase=0*)

Evaluate a chirp signal at time t.

A chirp signal is a frequency swept cosine wave.

$$a = \pi(f_1 - f_0)/t_1$$

$$b = 2\pi f_0$$

$$y = \cos\left(\pi\frac{f_1 - f_0}{t_1}t^2 + 2\pi f_0 t + \text{phase}\right)$$

Parameters

- **t** (*array*) – times at which to evaluate the chirp signal
- **f0** (*float*) – frequency at time t=0 (Hz)
- **t1** (*float*) – time t1
- **f1** (*float*) – frequency at time t=t1 (Hz)
- **form** (*str*) – shape of frequency sweep in ['linear', 'quadratic', 'logarithmic']
- **phase** (*float*) – phase shift at t=0

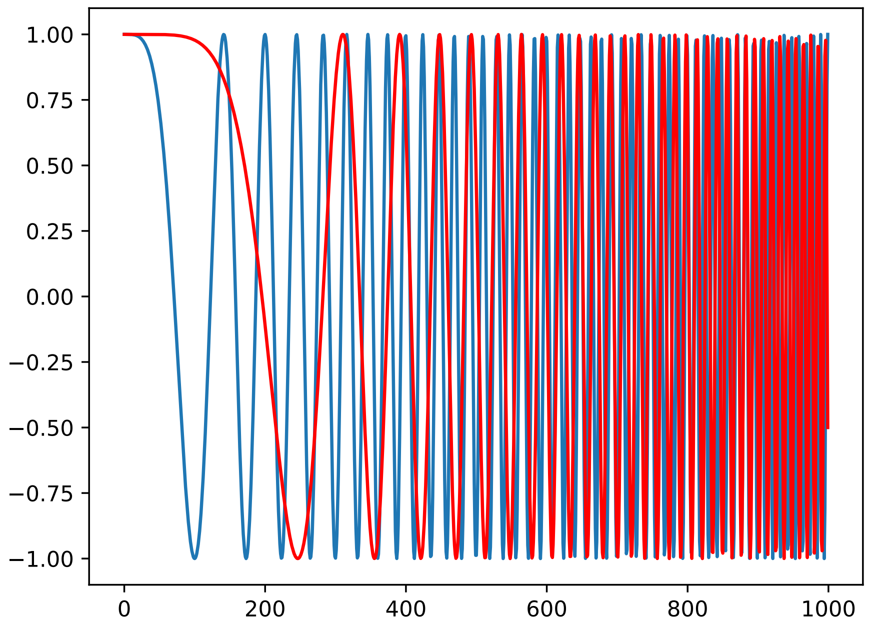
The parameter **form** can be:

- 'linear' $f(t) = (f_1 - f_0)(t/t_1) + f_0$
- 'quadratic' $f(t) = (f_1 - f_0)(t/t_1)^2 + f_0$

- 'logarithmic' $f(t) = (f_1 - f_0)^{(t/t_1)} + f_0$

Example:

```
from spectrum import chirp
from pylab import linspace, plot
t = linspace(0, 1, 1000)
y = chirp(t, form='linear')
plot(y)
y = chirp(t, form='quadratic')
plot(y, 'r')
```



mexican (*lb*, *ub*, *n*)

Generate the mexican hat wavelet

The Mexican wavelet is:

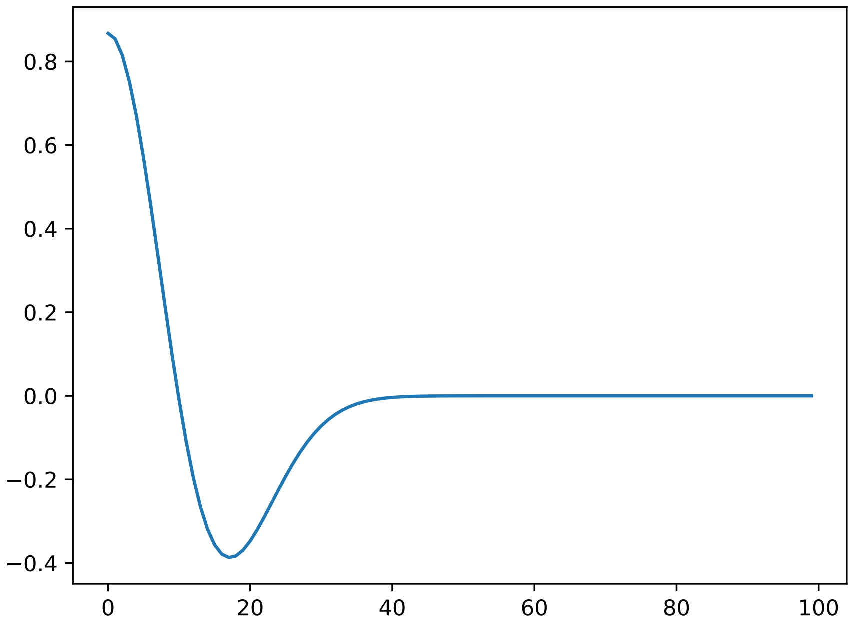
$$w[x] = \cos 5x \exp^{-x^2/2}$$

Parameters

- **lb** – lower bound
- **ub** – upper bound
- **n** (*int*) – waveform data samples

Returns the waveform

```
from spectrum import mexican
from pylab import plot
plot(mexican(0, 10, 100))
```



meyeraux (*x*)

Compute the Meyer auxiliary function

The Meyer function is

$$y = 35x^4 - 84x^5 + 70x^6 - 20x^7$$

Parameters **x** (*array*) –

Returns the waveform

```
from spectrum import meyeraux
from pylab import linspace, plot
t = linspace(0, 1, 1000)
plot(t, meyeraux(t))
```

5.5.7 Linear prediction

Linear prediction tools

References [Kay]

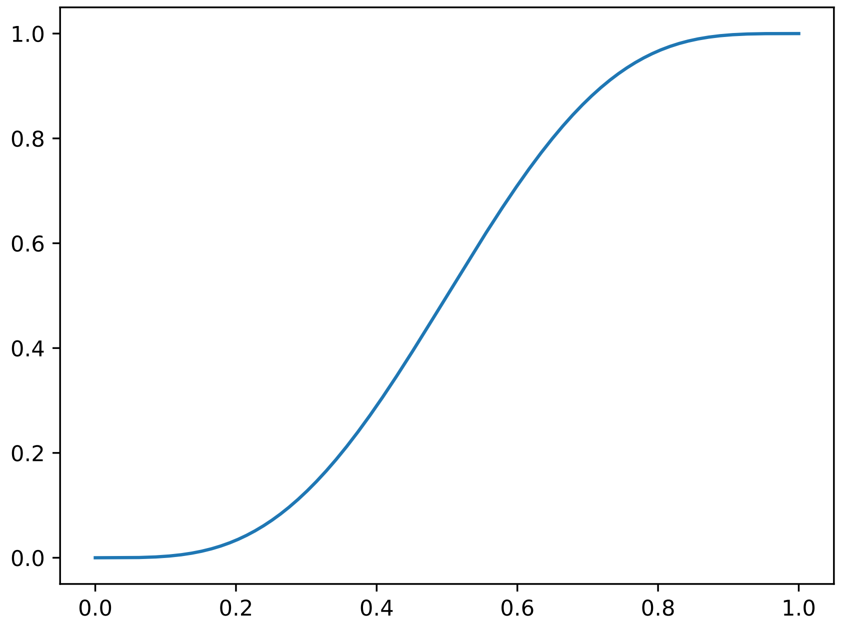
ac2poly (*data*)

Convert autocorrelation sequence to prediction polynomial

Parameters **data** (*array*) – input data (list or numpy.array)

Returns

- AR parameters
- noise variance



This is an alias to:

```
a, e, c = LEVINSON(data)
```

Example

```
>>> from spectrum import ac2poly
>>> from numpy import array
>>> r = [5, -2, 1.01]
>>> ar, e = ac2poly(r)
>>> ar
array([ 1.  ,  0.38, -0.05])
>>> e
4.1895000000000007
```

poly2ac (*poly*, *efinal*)

Convert prediction filter polynomial to autocorrelation sequence

Parameters

- **poly** (*array*) – the AR parameters
- **efinal** – an estimate of the final error

Returns the autocorrelation sequence in complex format.

```
>>> from numpy import array
>>> from spectrum import poly2ac
>>> poly = [ 1. ,  0.38 , -0.05]
>>> efinal = 4.1895
```

(continues on next page)

(continued from previous page)

```
>>> poly2ac(poly, efinal)
array([ 5.00+0.j, -2.00+0.j,  1.01-0.j])
```

ac2rc(*data*)

Convert autocorrelation sequence to reflection coefficients

Parameters **data** – an autorrelation vector

Returns the reflection coefficient and data[0]

This is an alias to:

```
a, e, c = LEVINSON(data)
c, data[0]
```

rc2poly(*kr*, *r0=None*)

convert reflection coefficients to prediction filter polynomial

Parameters **k** – reflection coefficients

rc2ac(*k*, *R0*)

Convert reflection coefficients to autocorrelation sequence.

Parameters

- **k** – reflection coefficients
- **R0** – zero-lag autocorrelation

Returns the autocorrelation sequence

See also:

ac2rc(), *poly2rc()*, *ac2poly()*, *poly2rc()*, *rc2poly()*.

is2rc(*inv_sin*)

Convert inverse sine parameters to reflection coefficients.

Parameters **inv_sin** – inverse sine parameters

Returns reflection coefficients

See also:

rc2is(), *poly2rc()*, *ac2rc()*, *lar2rc()*.

Reference J.R. Deller, J.G. Proakis, J.H.L. Hansen, "Discrete-Time Processing of Speech Signals", Prentice Hall, Section 7.4.5.

rc2is(*k*)

Convert reflection coefficients to inverse sine parameters.

Parameters **k** – reflection coefficients

Returns inverse sine parameters

See also:

is2rc(), *rc2poly()*, rc2acC(), *rc2lar()*.

Reference: J.R. Deller, J.G. Proakis, J.H.L. Hansen, "Discrete-Time Processing of Speech Signals", Prentice Hall, Section 7.4.5.

rc2lar(*k*)
Convert reflection coefficients to log area ratios.

Parameters **k** – reflection coefficients

Returns inverse sine parameters

The log area ratio is defined by G = log((1+k)/(1-k)) , where the K parameter is the reflection coefficient.

See also:

lar2rc(), *rc2poly()*, *rc2ac()*, rc2ic().

References [1] J. Makhoul, "Linear Prediction: A Tutorial Review," Proc. IEEE, Vol.63, No.4, pp.561-580, Apr 1975.

lar2rc(*g*)
Convert log area ratios to reflection coefficients.

Parameters **g** – log area ratios

Returns the reflection coefficients

References [1] J. Makhoul, "Linear Prediction: A Tutorial Review," Proc. IEEE, Vol.63, No.4, pp.561-580, Apr 1975.

poly2rc(*a*, *efinal*)
Convert prediction filter polynomial to reflection coefficients

Parameters

- **a** – AR parameters
- **efinal** –

lsf2poly(*lsf*)
Convert line spectral frequencies to prediction filter coefficients

returns a vector a containing the prediction filter coefficients from a vector lsf of line spectral frequencies.

```
>>> from spectrum import lsf2poly
>>> lsf = [0.7842 ,   1.5605  ,  1.8776 ,   1.8984,    2.3593]
>>> a = lsf2poly(lsf)
```

# array([1.00000000e+00, 6.14837835e-01, 9.89884967e-01, # 9.31594056e-05, 3.13713832e-03, -8.12002261e-03])

See also:

poly2lsf, rc2poly, ac2poly, rc2is

poly2lsf(*a*)
Prediction polynomial to line spectral frequencies.

converts the prediction polynomial specified by A, into the corresponding line spectral frequencies, LSF. normalizes the prediction polynomial by A(1).

```
>>> from spectrum import poly2lsf
>>> a = [1.0000,  0.6149, 0.9899, 0.0000 ,0.0031, -0.0082]
>>> lsf = poly2lsf(a)
>>> lsf =  array([0.7842, 1.5605, 1.8776, 1.8984, 2.3593])
```

See also:

lsf2poly, poly2rc, poly2qc, rc2is

5.6 Bibliography

For those interested, here are some books and articles used to design this library.

5.6.1 Books

5.6.2 Articles

- John Parker Burg (1968) “A new analysis technique for time series data”, NATO advanced study Institute on Signal Processing with Emphasis on Underwater Acoustics, Enschede, Netherlands, Aug. 12-23, 1968.
- Steven M. Kay and Stanley Lawrence Marple Jr.: “Spectrum analysis – a modern perspective”, Proceedings of the IEEE, Vol 69, pp 1380-1419, Nov., 1981
- Abd-Krim Seghouane and Maiza Bekara “A small sample model selection criterion based on Kullback’s symmetric divergence”, IEEE Transactions on Signal Processing, Vol. 52(12), pp 3314-3323, Dec. 2004

CHAPTER 6

ChangeLog Summary

6.1 Version 0.8 (2020)

- 0.8.0: Nov 2020
  - Fixed documentation related to https://github.com/cokelaer/spectrum/issues/57
  - Better documentation for pmtm. Also, following https://github.com/cokelaer/spectrum/issues/68 issue, add a warning to advice users to use Multitapering class instead of pmtm function to plot the results.

6.2 Version 0.7 (2019)

- 0.7.6: Jan 2019
  - Accepted 3 PR (typos in docs)
  - Fixed bug reported in https://github.com/cokelaer/spectrum/issues/62 (CORRELOGRAMPSD function)
- 0.7.5: Dec 2018
  - add tight_layout in window module. https://github.com/cokelaer/spectrum/issues/52
  - pull request accepted https://github.com/cokelaer/spectrum/pull/50 from anielsen001 contributor
  - double sqrt() into double (double);
  - Fix https://github.com/cokelaer/spectrum/issues/52 (better plotting layout)
  - Update doc with dynamic carousel
  - BUG fix https://github.com/cokelaer/spectrum/issues/54 reported https://github.com/alfredo-f user
- 0.7.4: (Aug 2018)
  - Fixed issue https://github.com/cokelaer/spectrum/issues/47
- 0.7.3: (jan 2018)

  - Just a version update to push on pypi this bug fix: https://github.com/cokelaer/spectrum/commit/4b9cf4f08f090cdc36fd6cae3f4c87b5f5311e45

- 0.7.2:
  - **NEWS:**
    - add Taylor windows
- 0.7.1:
  - **NEWS:**
    - add io module with readwav function
    - add spectrogram module
    - add a wav data file example (DOLPHIN.WAV) for example
    - add MultiTapering class (calls pmtm)
  - **BUG Fixes:**
    - 2D case for speriodogram should work now
- 0.7.0:
  - **BUG fixes:**
    - Fix https://github.com/cokelaer/spectrum/issues/38 in pburg to have the correct amplitude like in octave. fixed by removing the call to scale() function
    - Similarly all other parametric methods have been changed by adding the scale_by_freq argument where missing
  - **Changes:**
    - remove cohere module

6.3 Version 0.6

- 0.6.8:
  - Fix the MANIFEST
- 0.6.7:
  - refactored the requirements files (add a requirements-dev.txt) and update the documentation (installation) accordingly
  - **BUG fixes:**
    - correlogram: real-data case had the data flipped
    - pmusic/pev: real-data case had the data flipped
    - fix the AKICc criteria code
  - **Updates:**
    - pmusic/pev: add the threshold and criteria arguments
    - more tests for the criteria and eigenfre modules
  - **Changes:**
    - Spectrum class: remove _correlogram method (use pcorrelogram instead)

- 0.6.6:
  - integration pull request https://github.com/cokelaer/spectrum/pull/29 from moritz-ritter to allow spectrum to run indepdently of matplotlib (for server head-less integration)
- 0.6.5:
  - minor updates to port spectrum on travis
- 0.6.4:
  - CHANGES: the bug reported in https://github.com/cokelaer/spectrum/issues/24 is obsolet for the reported module (pburg), which was fixed earlier but the issue was fixed in other module such as psd, parma, correlog
  - add LICENSE file
  - fix warning in cpp code (adding void in func() prototypes)
- 0.6.3:
  - CHANGES: portage nosetests suite to pytest
  - BUG Fixes:
  - Fix issues https://github.com/cokelaer/spectrum/issues/21 and https://github.com/cokelaer/spectrum/issues/20 mostly related to compatibility with newest numpy version (1.12)
- **0.6.2:**
  - **Bug Fixes:**
    - Issue #11: fixes loading mydpss library using numpy helper
    - Issue #12: Allow loading the shared library for frozen projects. Tested with py2exe.
  - **Changes:**
    - pmtm returns Sk_complex, weights and eigenvalues instead of just Sk
- **0.6.1:**
  - **BUG fixes**
    - Issue #5 in pyule sampling not initialised is now fixed
- **0.6.0:**
  - Code moved to github
  - plots accept the ax argument in psd module. It is a bit of a hack but seems to work.

6.4 Sept 2012

- **0.5.5:**
  - fix name of the libraries for mac and windows
  - change setup to manage version properly.

6.5 March 2012

- 0.5.3: add poly2lsf and lsf2poly, add tests, fix bug related to compilation of mydpss.cc
- 0.5.2: add pmtm

6.6 February 2012

- 0.5.1: add dpss wtapering windows
- **0.5.0:**
    - NPSD replaced by NFFT (qlso not correct for ARMA methods that do not have NFFT since not fourier)
    - Correlogram replaced by pcorrelogram
    - more consistent function and class naming convention
    - Update the entire documentation.
- 0.4.6: fixed pylab_periodogram, documentation (installation)

6.7 January 2012

- 0.4.5: start to play with Pypi

6.8 October 2011

- 0.4.4: Start to provide the library on the web www.assembla.com

6.9 May 2011

- 0.4.3: `spectrum.periodogram.pdaniell()` implemented

6.10 April 2011

- 0.4.2: pcovar implemented
- 0.4.1: pmodcovar implemented
- 0.4.0: arcovar and modcovar "simplified" version. Documentation updated (tutorial, spectral_estimation, quick start...)
- 0.3.19: add linear_prediction module with codecs (eg. ac2poly, poly2rc....)
- 0.3.18 fix bug in levinson (Real data case only) and add ac2poly function.
- 0.3.17: validation of the modcovar algorithm versus the new arcovar_simplified function.

- 0.3.16: add a simplified version of arcovar called arcovar_simplified. It is 10 times faster and with a different algorithm provides the same results as arcoar, which validates the two codes!
- 0.3.15: add corrmtx function. Tested it within music algorithm
- 0.3.14: cleanup the eigen and music methods by moving the automatic order selection outside the functions.
- 0.3.13: Add AIC and MDL criteria to deal with automatic eigen values selection in pmusic and pev
- 0.3.12: test and validate the pmusic and pev pseudo spectrum.
- 0.3.11: burg and pburg finalised
- 0.3.10: tools module cleanup and finalised
- 0.3.9: ma fully checked and add pma validated
- 0.3.8: minvar fully checked and add pminvar
- 0.3.7: aryule fully checked and add pyule
- 0.3.6: Speed up by 3 the ARMPSD (renamed to arma2psd)
- 0.3.5: refactoring
- 0.3.4: fix all tests and doctests
- 0.3.3: function Daniell's periodogram implemented in module periodogram
- 0.3.2: Create class MovingAverage, pburg, pARMA, Correlogram, Periodogram, Minvar, pma
- 0.3.1: Cleanup MA, ARMA, BURG, MINVAR
- 0.3.0: Create an ABC class Spectrum, a FourierSpectrum and ParametricSpectrum.
- 0.2.4: Finalise doc/test of the testdata module
- 0.2.3: define a PSD class
- 0.2.2: cleanup cholesky.py
- 0.2.1: a new sphinx layout,
- 0.2.0: correlogram.py, correlation.py, levinson.py fully completed

6.11 March 2011

- **31 March:**
  - finalise a criteria class for AIC, FPE criteria. Incorporated it in arburg
- **28th March:**
  - First version of `arcov()`, `aryule()` and `arburg()`
  - add many windows (parzen, flattop, . . .).
- **22th March 2011:**
  - put this doc online on thomas-cokelaer.info (fixed main links)
- **21th March 2011:**
  - create psd.py defines useful class to manage Spectrum/plot
  - periodogram.py has a simple periodogram implementation equivalent to psd in pylab without overlaping.

- **7th March 2011:**
  - add periodogram module
  - fix ARMA method in arma module
- **4th March 2011:**
  - Create first revision of spectrum package

CHAPTER 7

Contributions

7.1 Contributors

Here is a non-exhaustive list of contributors. If you have contributed and are not listed, please update the AUTHORS.rst file in the source repository.

- Thomas Cokelaer (main author)
- https://github.com/juhasch
- https://github.com/anielsen001
- https://github.com/carlkl
- https://github.com/gozzilli
- https://github.com/wanglongqi
- https://github.com/alfredo-f

–

For an up-to-date page of source code contributors, please visit the github repository pages.

7.2 How to contribute ?

If you have found a bug, have issues or contributions, please join the development on https://github.com/cokelaer/spectrum.

7.3 Some notebooks

- http://nbviewer.ipython.org/gist/juhasch/5182528

CHAPTER 8

License

Spectrum is released under a BSD3 license

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Bibliography

[Hayes] Hayes, M.H. Statistical Digital Signal Processing and Modeling. New York: John Wiley & Sons, 1996.

[Kay] Kay, S.M. Modern Spectral Estimation. Englewood Cliffs, NJ: Prentice Hall, 1988.

[Marple] Marple, S.L. Digital Spectral Analysis. Englewood Cliffs, NJ: Prentice Hall, 1987.

[Orfanidis] Orfanidis, S.J. Introduction to Signal Processing. Upper Saddle River, NJ: Prentice Hall, 1996.

[Percival] Percival, D.B., and A.T. Walden. Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques. Cambridge: Cambridge University Press, 1993.

[Proakis] Proakis, J.G., and D.G. Manolakis. Digital Signal Processing: Principles, Algorithms, and Applications. Englewood Cliffs, NJ: Prentice Hall, 1996.

[Stoica] Stoica, P., and R. Moses. Introduction to Spectral Analysis. Upper Saddle River, NJ: Prentice Hall, 1997.

[octave] octave software

[Harris] Harris, F.J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." Proceedings of the IEEE. Vol. 66, No. 1 (January 1978).

[Nuttall] Nuttall, Albert H. "Some Windows with Very Good Sidelobe Behavior." IEEE Transactions on Acoustics, Speech, and Signal Processing. Vol. ASSP-29 (February 1981). pp. 84-91.

[Wax] Wax, M. and Kailath, T. Detection of signals by information Theoretic criteria, IEEE Trans Acoust. Speech Signal Process, vol ASSP-33, pp 387-392, 1985.

[Welch] Welch, P.D. "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms." IEEE Trans. Audio Electroacoust. Vol. AU-15 (June 1967). Pgs.70-73.

Python Module Index

Index