spectralAnalysisTutorial2019

July 11, 2019

1 Tutorial : Spectral analysis of stochastic processes

- The material for this tutorial is available at http://bit.ly/1WMbAOX, make sure to put the lib subdirectory in your path with a line like this:

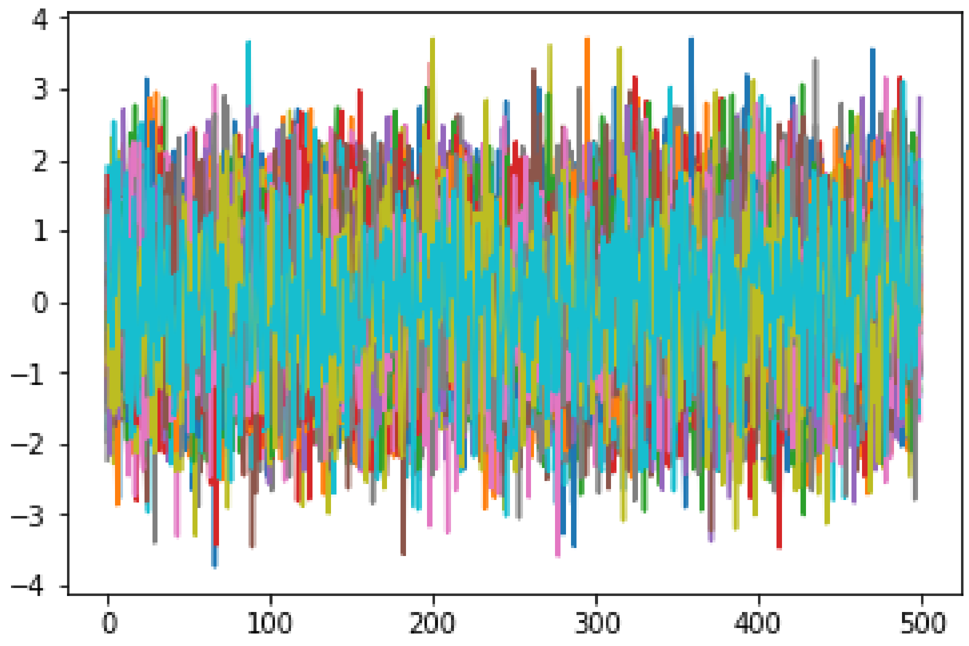
```
In [1]: import numpy as np
        from scipy.signal import welch,butter,lfilter
```

The following command triggers inline plot

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

We first generate N time points for K realizations of an i.i.d. gaussian noise with zero mean and variance one.

```
In [4]: # Duration of the signals
        N = 500
        # Number of realizations
        K = 40
        x = np.random.randn(N,K)
        plt.plot(x)
        plt.show()
```

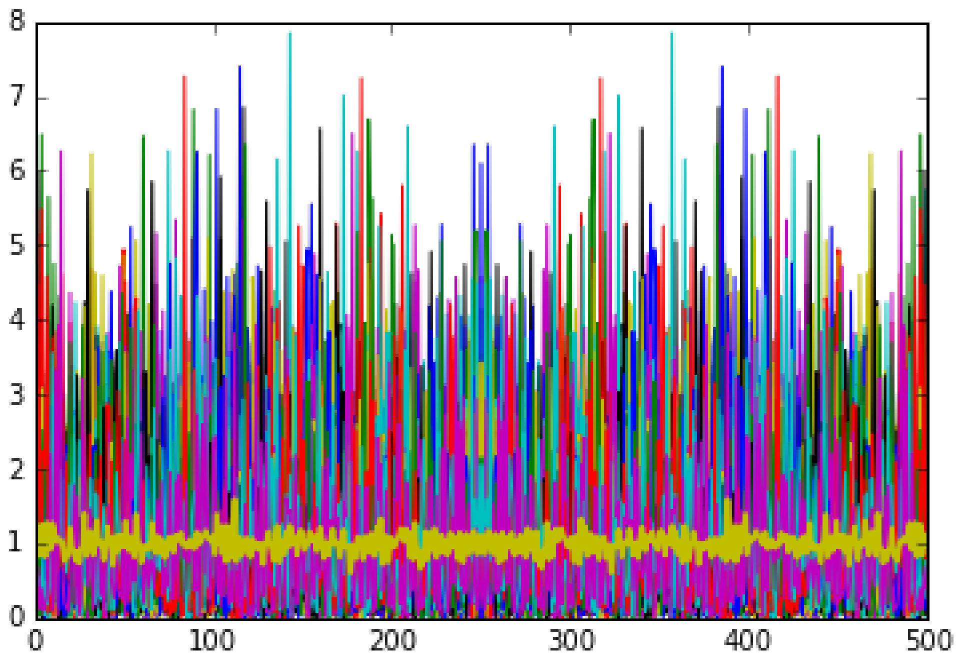


Let us plot the Periodogram

$$P_N(x) = \frac{1}{N}|F_N x|^2$$

of these K realizations.

```
In [106]: plt.plot(np.abs(np.fft.fft(x,axis=0))**2/N)
          plt.plot(np.mean(np.abs(np.fft.fft(x,axis=0))**2,axis=1)/N,linewidth=2)
          plt.show()
```



We see that the periodogram is not a good estimate of the Power Spectral Density, because its values fluctuate a lot. However, we notice by averaging the periodogram of several realizations that the empirical mean gets close to the expected constant PSD, this constant being the variance of the signal, i.e. one.

In order to have a better estimate of the PSD, we use the Welch periodogram.

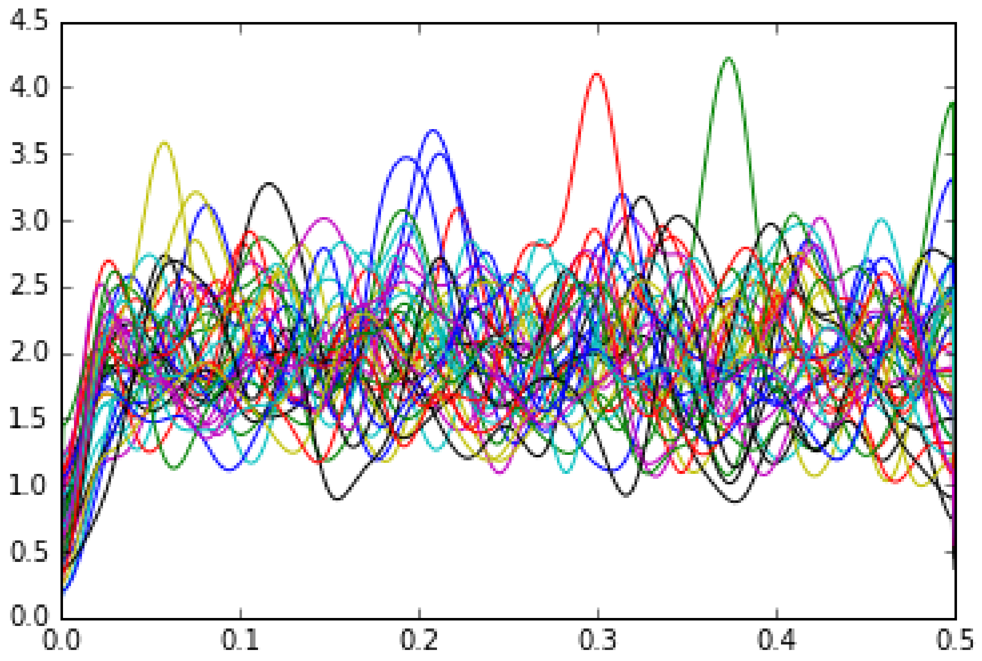
```
In [107]: help(welch)
```

```
Help on function welch in module scipy.signal.spectral:

welch(x, fs=1.0, window='hanning', nperseg=256, noverlap=None, nfft=None, detrend='constant', 
    Estimate power spectral density using Welch's method.

    Welch's method [1]_ computes an estimate of the power spectral density
    by dividing the data into overlapping segments, computing a modified
    periodogram for each segment and averaging the periodograms.

    Parameters
    ----------
    x : array_like
        Time series of measurement values
    fs : float, optional
        Sampling frequency of the `x` time series. Defaults to 1.0.
    window : str or tuple or array_like, optional
        Desired window to use. See `get_window` for a list of windows and
        required parameters. If `window` is array_like it will be used
        directly as the window and its length will be used for nperseg.
        Defaults to 'hanning'.
    nperseg : int, optional
        Length of each segment.  Defaults to 256.
    noverlap : int, optional
        Number of points to overlap between segments. If None,
        ``noverlap = nperseg // 2``.  Defaults to None.
    nfft : int, optional
        Length of the FFT used, if a zero padded FFT is desired.  If None,
        the FFT length is `nperseg`. Defaults to None.
    detrend : str or function or False, optional
        Specifies how to detrend each segment. If `detrend` is a string,
        it is passed as the ``type`` argument to `detrend`.  If it is a
        function, it takes a segment and returns a detrended segment.
        If `detrend` is False, no detrending is done.  Defaults to 'constant'.
    return_onesided : bool, optional
        If True, return a one-sided spectrum for real data. If False return
        a two-sided spectrum. Note that for complex data, a two-sided
        spectrum is always returned.
    scaling : { 'density', 'spectrum' }, optional
        Selects between computing the power spectral density ('density')
        where `Pxx` has units of V**2/Hz and computing the power spectrum
        ('spectrum') where `Pxx` has units of V**2, if `x` is measured in V
        and fs is measured in Hz.  Defaults to 'density'
    axis : int, optional
        Axis along which the periodogram is computed; the default is over
        the last axis (i.e. ``axis=-1``).

    Returns
    -------
    f : ndarray
        Array of sample frequencies.
    Pxx : ndarray
        Power spectral density or power spectrum of x.

    See Also
    --------
    periodogram: Simple, optionally modified periodogram
    lombscargle: Lomb-Scargle periodogram for unevenly sampled data

    Notes
    -----
    An appropriate amount of overlap will depend on the choice of window
    and on your requirements.  For the default 'hanning' window an
    overlap of 50% is a reasonable trade off between accurately estimating
    the signal power, while not over counting any of the data.  Narrower
    windows may require a larger overlap.
```

```
If `noverlap` is 0, this method is equivalent to Bartlett's method [2]_.

.. versionadded:: 0.12.0

References
----------
.. [1] P. Welch, "The use of the fast Fourier transform for the
       estimation of power spectra: A method based on time averaging
       over short, modified periodograms", IEEE Trans. Audio
       Electroacoust. vol. 15, pp. 70-73, 1967.
.. [2] M.S. Bartlett, "Periodogram Analysis and Continuous Spectra",
       Biometrika, vol. 37, pp. 1-16, 1950.

Examples
--------
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
>>> np.random.seed(1234)

Generate a test signal, a 2 Vrms sine wave at 1234 Hz, corrupted by
0.001 V**2/Hz of white noise sampled at 10 kHz.

>>> fs = 10e3
>>> N = 1e5
>>> amp = 2*np.sqrt(2)
>>> freq = 1234.0
>>> noise_power = 0.001 * fs / 2
>>> time = np.arange(N) / fs
>>> x = amp*np.sin(2*np.pi*freq*time)
>>> x += np.random.normal(scale=np.sqrt(noise_power), size=time.shape)

Compute and plot the power spectral density.

>>> f, Pxx_den = signal.welch(x, fs, nperseg=1024)
>>> plt.semilogy(f, Pxx_den)
>>> plt.ylim([0.5e-3, 1])
>>> plt.xlabel('frequency [Hz]')
>>> plt.ylabel('PSD [V**2/Hz]')
>>> plt.show()

If we average the last half of the spectral density, to exclude the
peak, we can recover the noise power on the signal.

>>> np.mean(Pxx_den[256:])
0.0009924865443739191

Now compute and plot the power spectrum.
```

```
>>> f, Pxx_spec = signal.welch(x, fs, 'flattop', 1024, scaling='spectrum')
>>> plt.figure()
>>> plt.semilogy(f, np.sqrt(Pxx_spec))
>>> plt.xlabel('frequency [Hz]')
>>> plt.ylabel('Linear spectrum [V RMS]')
>>> plt.show()

The peak height in the power spectrum is an estimate of the RMS amplitude.

>>> np.sqrt(Pxx_spec.max())
2.0077340678640727
```

```
In [115]: [f,Pxx] = welch(x,axis=0,nperseg=50,nfft=1024)
          plt.plot(f,Pxx)
          #plt.ylim([0, 5])
          plt.show()
          Pxx.shape
```

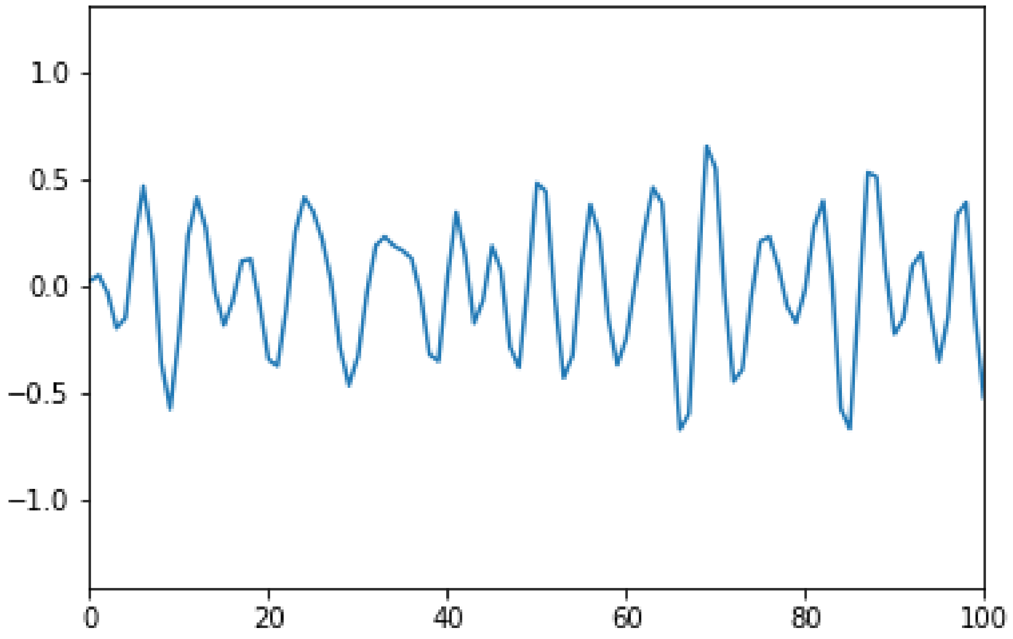


```
Out[115]: (513, 40)
```

The result, using the default parameters of pwelch, clearly reduces the fluctuations with respect to the periodogram estimate.

To better understand how to use pwelch, we now generate a *colored* signal by filtering a white noise using a butterworth band-pass filter. Note that when we plot the realizations of this signal, we observe oscillatory fluctuations reflecting the non-zero auto-correlation for non-zero lags.

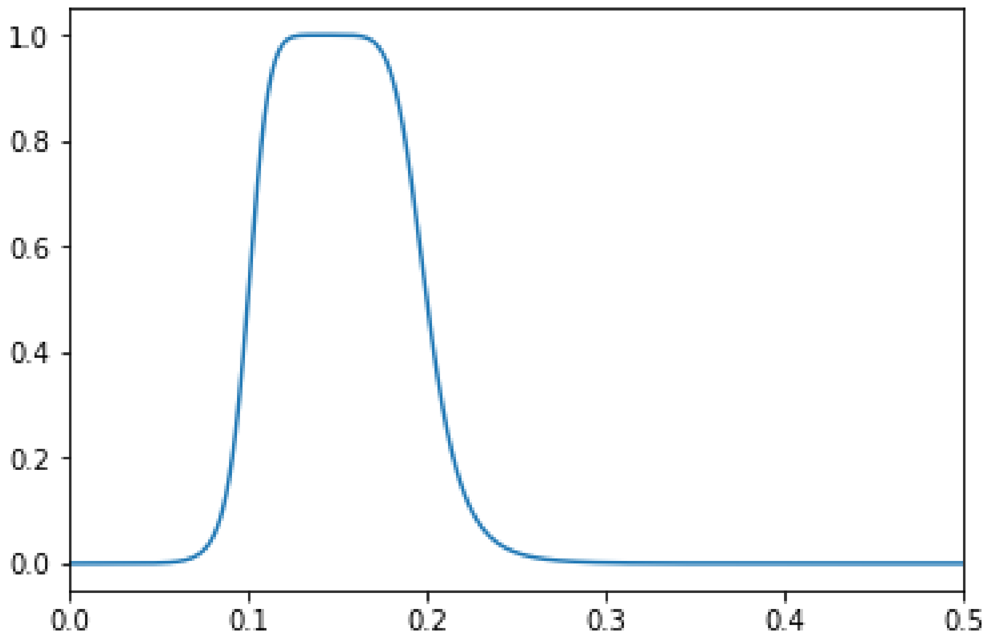
```
In [9]: [brd,ard] = butter(3,np.array([.2, .4]),'bandpass')
        y=lfilter(brd,ard,x,axis=0)
        plt.plot(y[:,4])
        plt.xlim([0, 100])
        plt.show()
```



By computing the impulse response of the filter h, we can infer the PSD of the filtered noise as

$$S_y(\nu) = |h(\nu)|^2$$

```
In [10]: dirac = np.zeros([10000,1])
         dirac[0] = 1
         impresp = lfilter(brd,ard,dirac,axis=0);
         irFreqAx = np.arange(len(impresp))/len(impresp);
         plt.plot(irFreqAx,np.abs(np.fft.fft(impresp,axis=0))**2)
         plt.xlim([0, .5])
         plt.show()
```

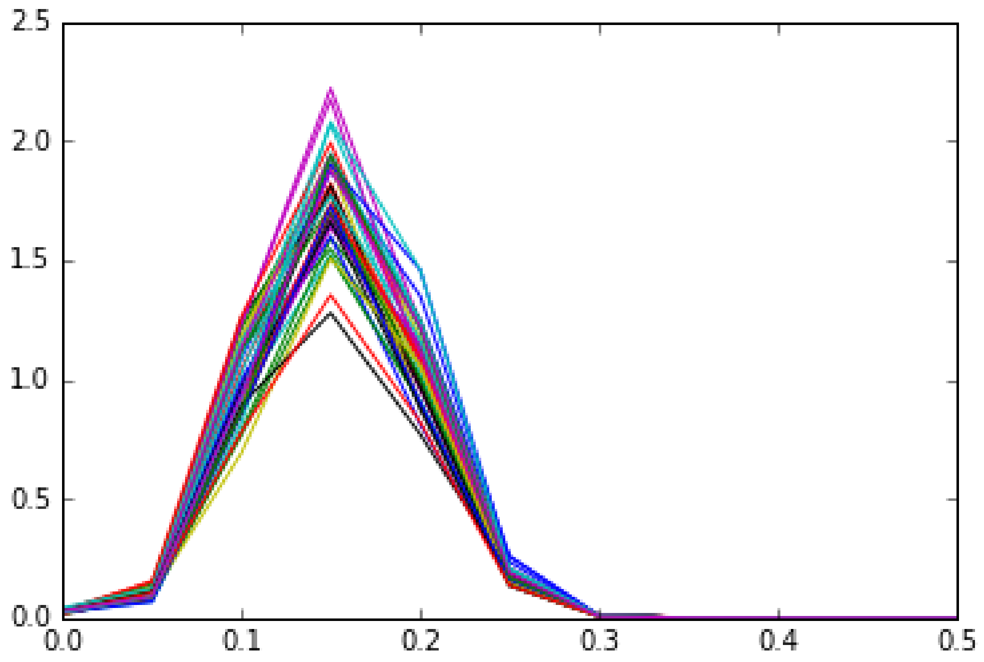


Now we try to estimate the PSD using pwelch. The main parameter of pwelch that we can modulate to affect the estimation procedure is the length of the time window *winLength* that is used to compute individual periodograms before averaging them.

The smaller the length of the time window, the coarser is the frequency resolution of the estimate. The can is due to the limitation of the DTF that is applied on each time window, and whose frequency resolution is typically $\frac{1}{winLength}$, i.e. the grid step of the DFT. On the other hand, dividing the time interval using smaller length time windows leads to more time windows, which lead to a greater reduction of the fluctuations of the estimator after averaging the periodograms.

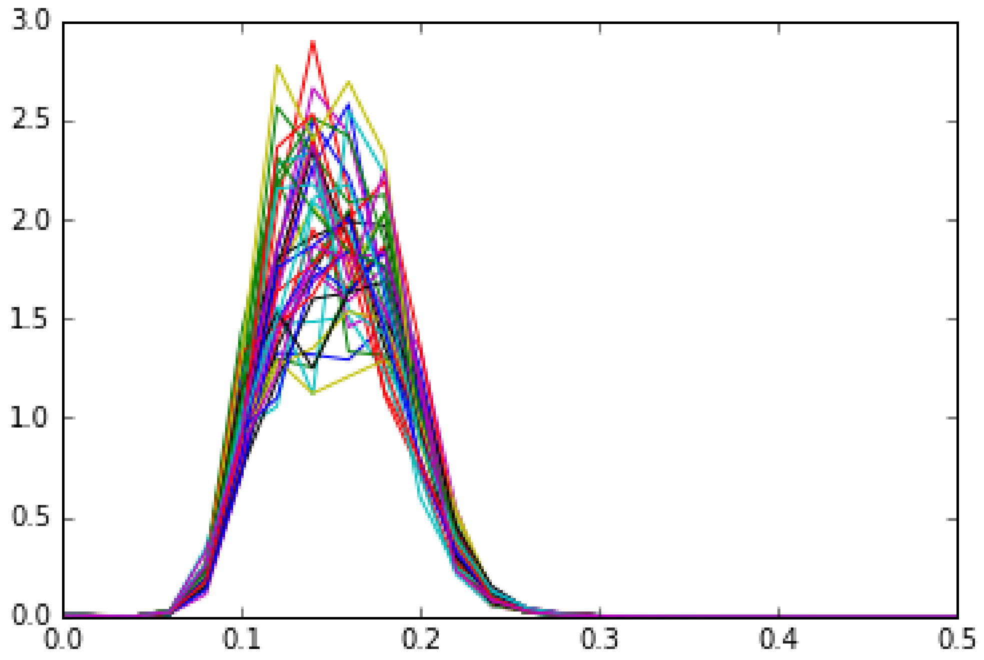
We first use a large time window.

```
In [131]: [freq,pwy] = welch(y,axis=0,nperseg=20);
          plt.plot(freq,pwy)
          plt.show()
```



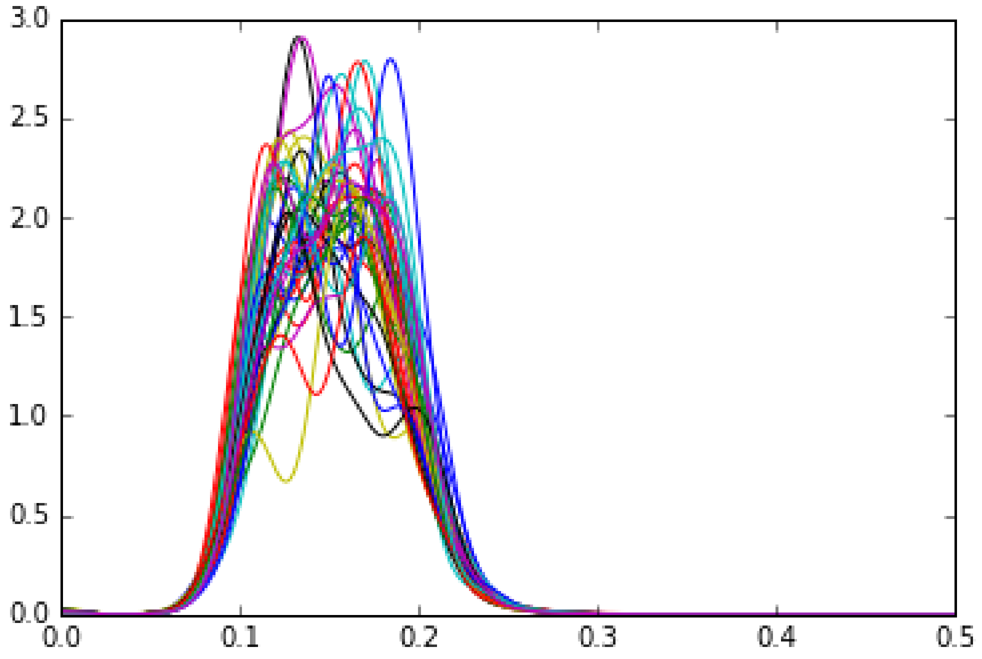
The fluctuations are still large. Now we reduce the length of the time window to 50 points.

```
In [46]: [freq,pwy] = welch(y,axis=0,nperseg=50);
         plt.plot(freq,pwy)
         plt.show()
```



We see that the fluctuations are reduced and the overall shape of the PSD is well captured. Next, we reduce again the window size.

```
In [133]: [freq,pwy] = welch(y,axis=0,nperseg=50,nfft=1024);
          plt.plot(freq,pwy)
          plt.show()
```



We see now that reducing too much the window size results in a poor frequency resolution, such that the shape of the PSD is not well captured anymore.

In order to choose the optimal parameters of the Welch periodogram, we thus need to have a prior knowledge of the frequency resultion of the PSD we are trying to estimate. The inverse of the frequency resolution provides a minimum value for the window length to be used. Ultimately, reaching a satifactory tradeoff between good frequency resolution and small fluctuations may require to acquire more time points.