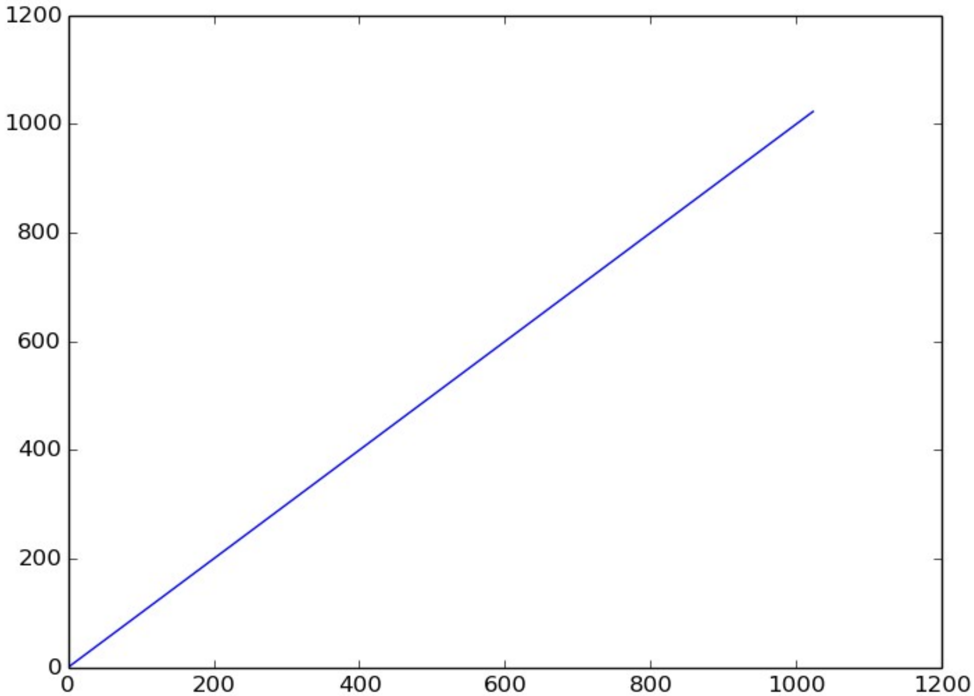
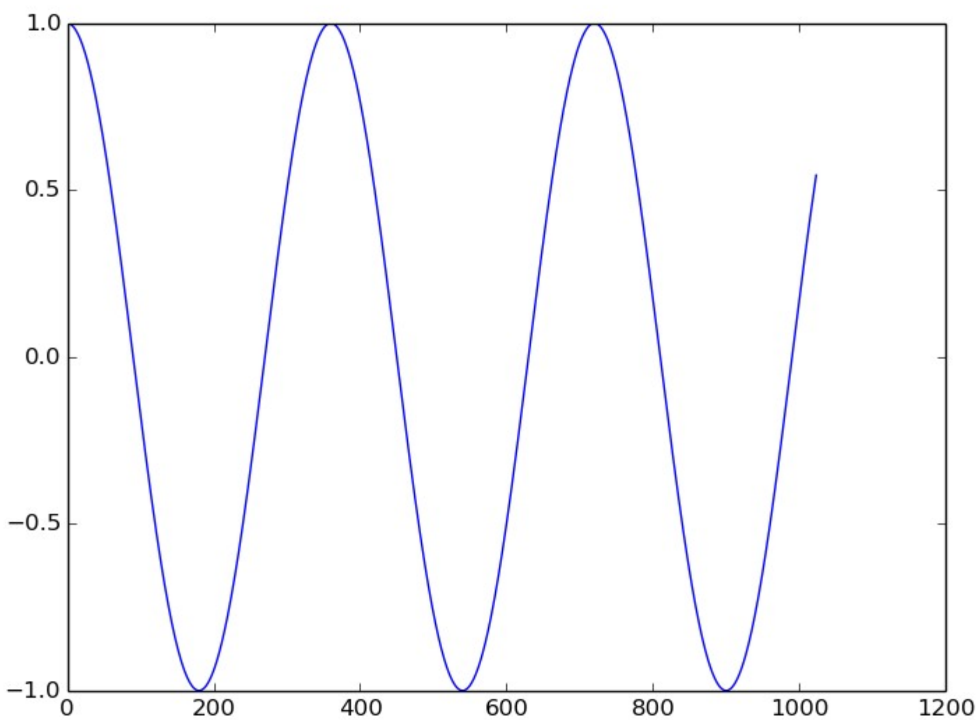
H. Altug Yildirim
090100252

Midterm Signal Process
Butterworth Filter

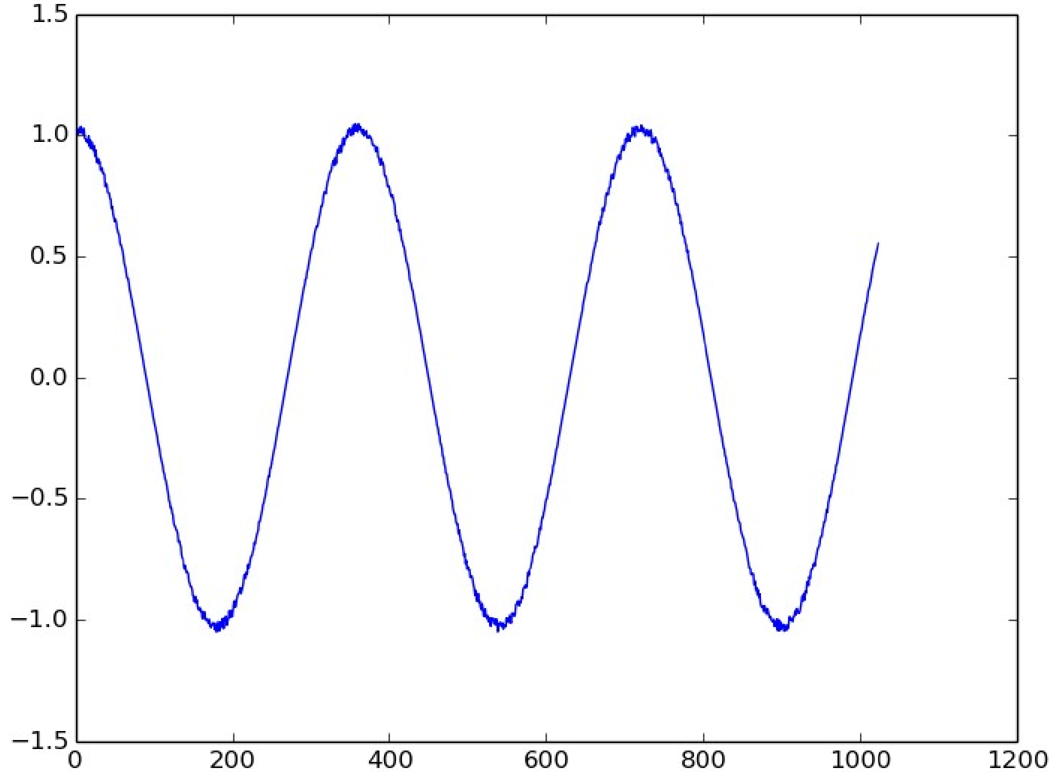
First the sample rate is stated as 1024(it needs to be even for the reasons of using discrete fourier transform.) Then time domain created:



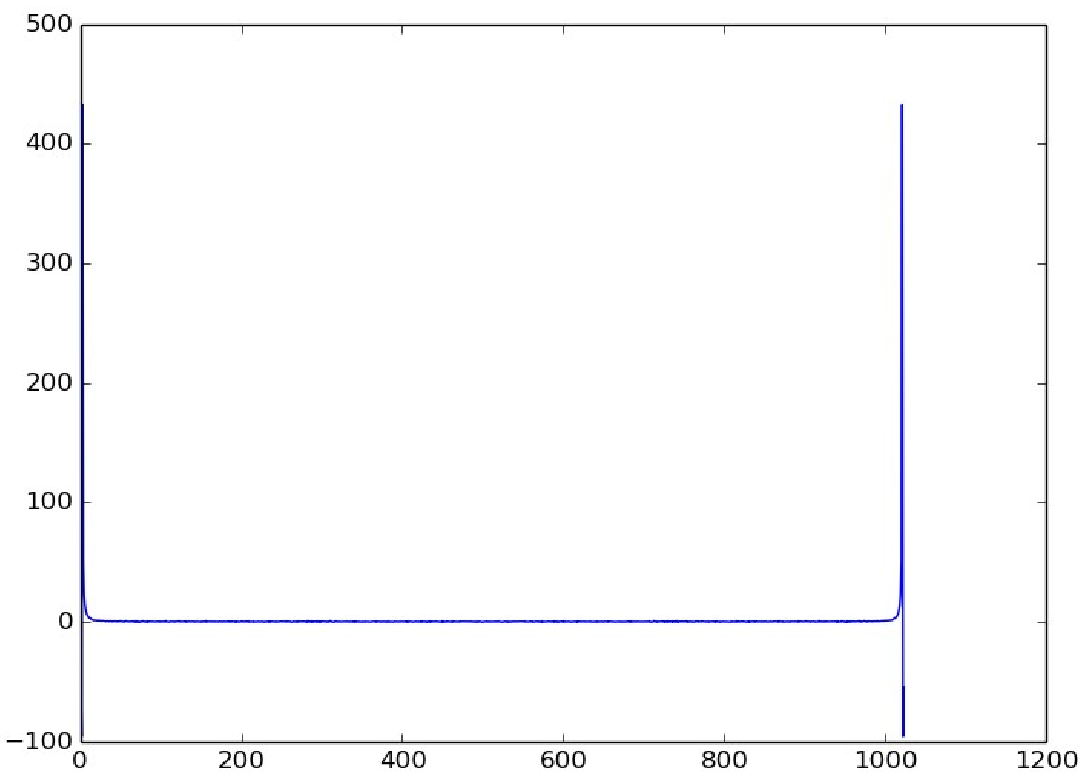
Then a cosine signal is created:



Then noise added to signal using random functions:



Then taking fast fourier transform of the signal. With the argument of 1024.



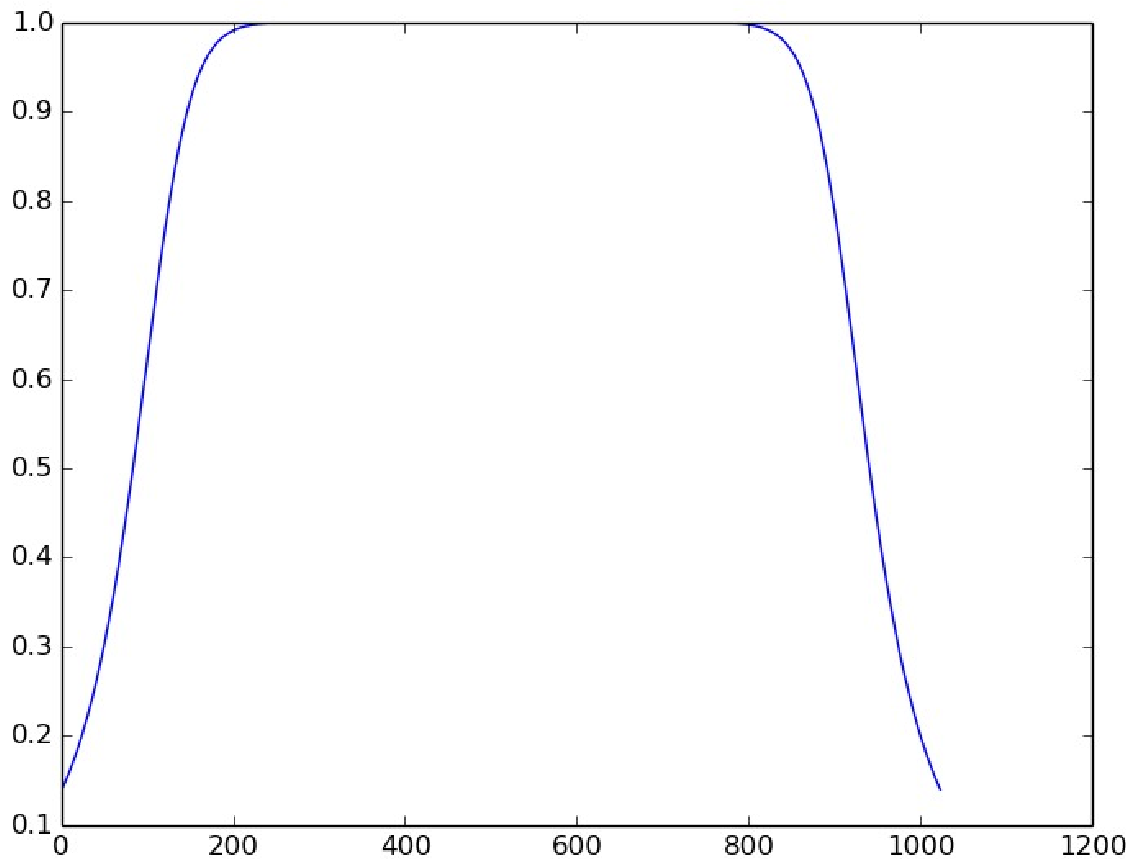
Then we will use the butterworth filter.

$$G=\frac{1}{\sqrt{(1+(f/f_c)^{2n})}}$$

But in order to not discriminate against the negative frequencies we have to shift this from 0,1024 to -512,512. To do this we subtract the nyquist frequency, which we find with sample rate divide by 2. So our signal happens to be:

$$G=\frac{1}{\sqrt{(1+((f-nyquist)/f_c)^{2n})}}$$

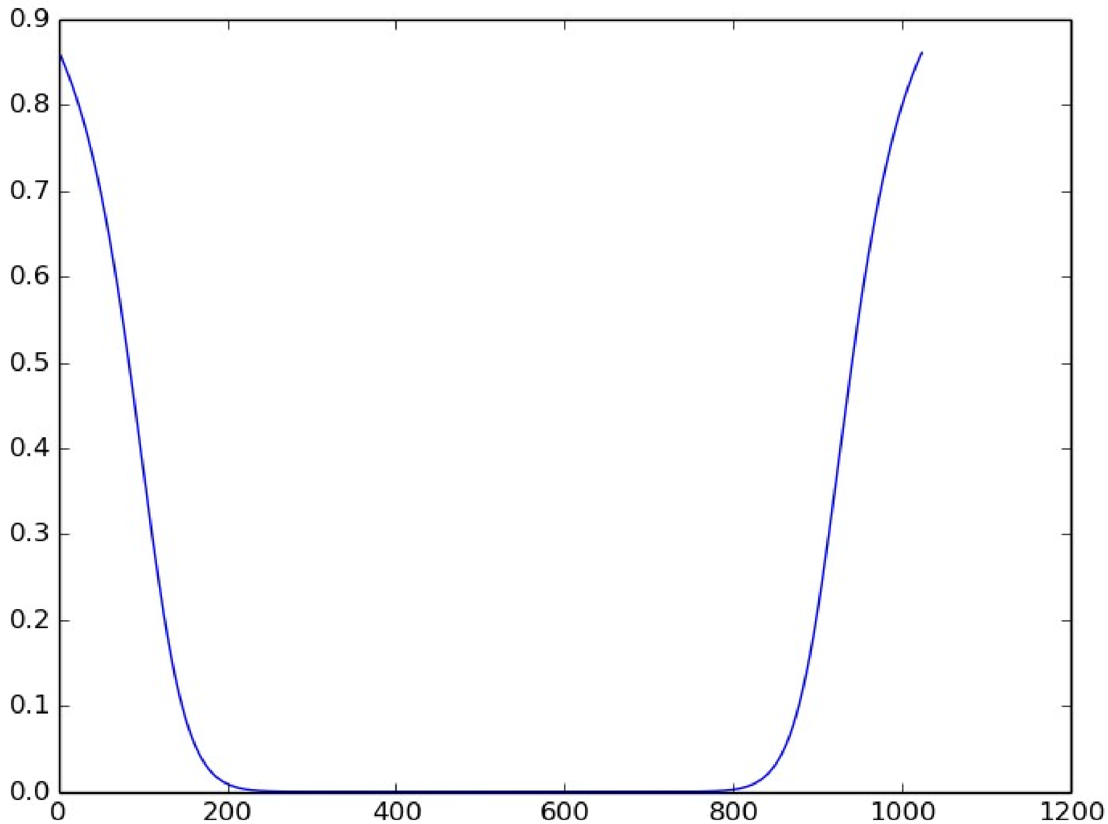
When we plot this:



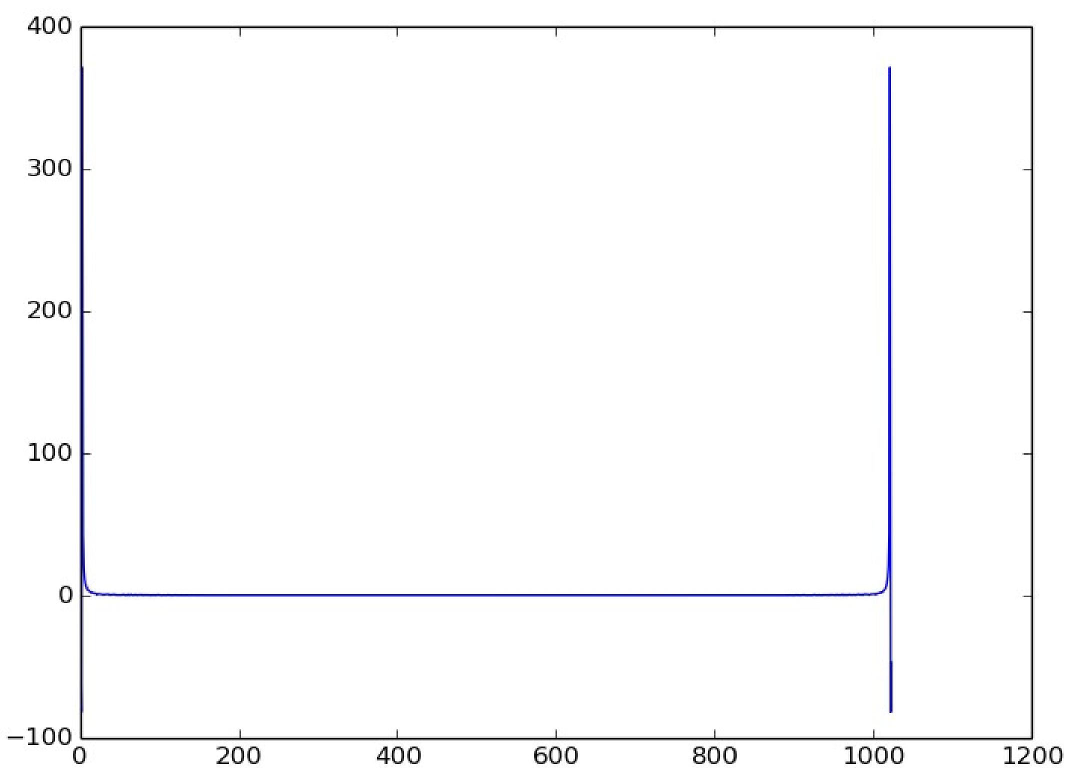
But this filter is not true for our signal because the needed frequencies for our signal is around the 0 frequency. To achieve this we have to invert this symmetrical to x-axis:

$$G=1-\frac{1}{\sqrt{(1+((f-nyquist)/f_c)^{2n})}}$$

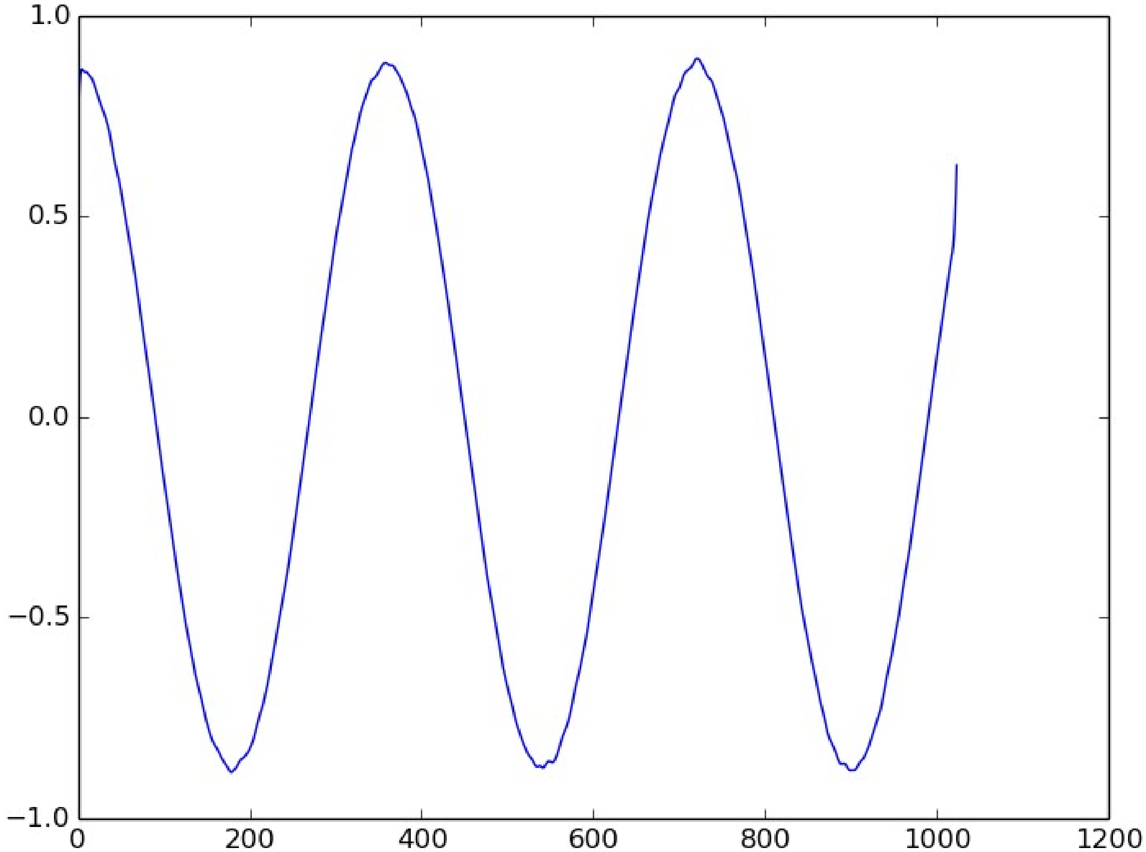
The order in use in here is 8 and the cut-off frequency is 400. Plotting this:



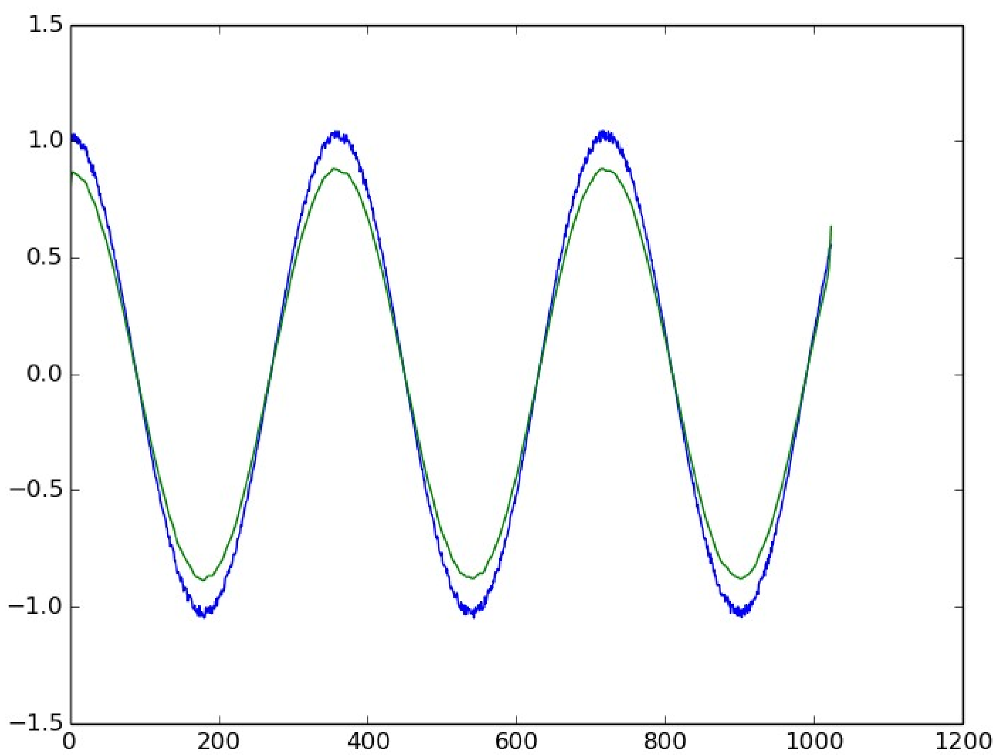
We apply this to our signal in frequency domain:



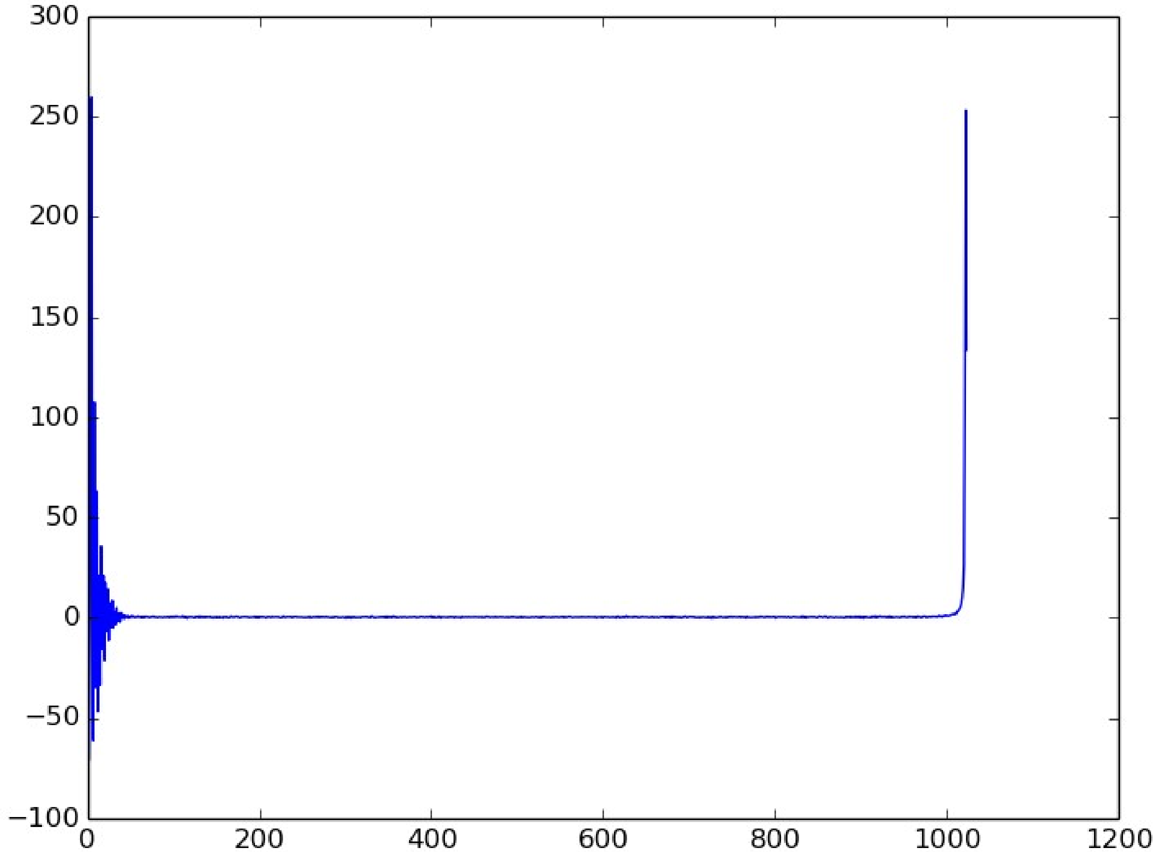
In graph it can be seen that noisy parts are gone with the filter. So taking the inverse fast fourier transform of the frequency domain signal we achieve:



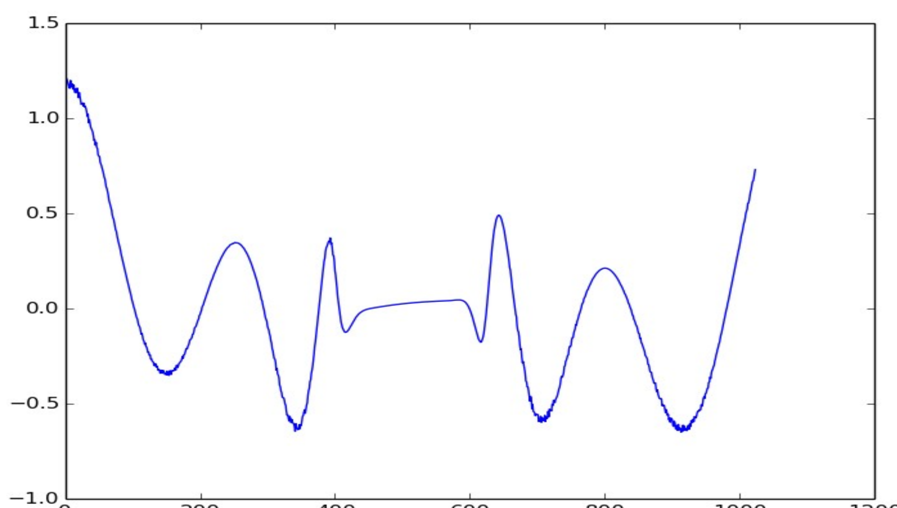
To compare the input signal and filtered signal:



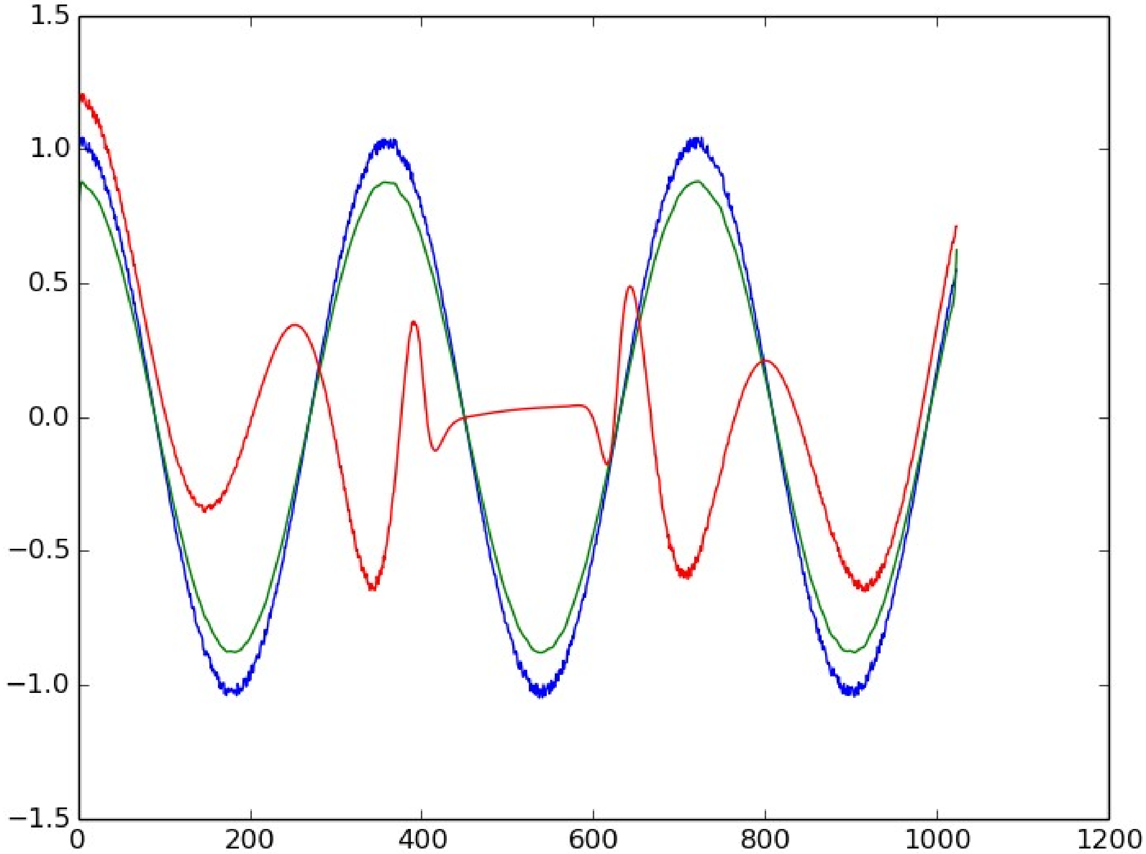
Additional Source: Also in order to have a reference I try to compare my results with the ready-to-use function of the SciPy library of the python but the results seems to be not very accurate. The frequency domain when we take use the function:



When we inverse fourier transform this:



Which seems very inaccurate. Comparing with the input signal, filtered signals with written filter and ready-to-use filter:



Our Code:

```
#libraries that needed for butterworth filter
from math import *
import numpy as np
from pylab import *
import random as ran
from scipy.signal import butter, lfilter

#Low-Pass Frequency Response Function
def lowpass_signal_func(t,frequency_domain_signal,f_c, nyq, n):
    h_lowpass=np.zeros(len(t))
    lowpass_signal=[]
    for i in range(len(frequency_domain_signal)):
        h_lowpass[i]=1-sqrt(1./(1+((i-nyq)/(f_c))**(2*n)))
        lowpass_signal.append(h_lowpass[i]*frequency_domain_signal[i])
    return lowpass_signal, h_lowpass

def scipy_lowpass_filter(frequency_domain_signal,f_c,nyq,n):
    cut=f_c/nyq
    b,a=butter(n,cut,btype='low')
    y=lfilter(b,a,frequency_domain_signal)
    return y


#Sampling Rate
f_s=1024.
#Nyquist Frequency
nyq=0.5 * f_s
#Generating Time Domain
t=linspace(0, f_s,f_s,endpoint=False)

#generating a clean signal
clean_signal = cos(t*pi/180)

#adding noise to clean signal
noise_signal=clean_signal
for i in range(len(clean_signal)):
    noise_signal[i]=noise_signal[i]*ran.uniform(1,1.05)

#Taking the Fast Fourier Transform of the signal
frequency_domain_signal=np.fft.fft(noise_signal,1024)

#Butterworth Filter
#Cutoff Frequency
f_c=400
#Order
n=8
frequency_domain_signal_filtered,G=lowpass_signal_func(t,frequency_domain_signal,f_c,nyq,n)
frequency_domain_signal_filtered_ready=scipy_lowpass_filter(frequency_domain_signal,f_c,nyq,n)
```

```
#Inverse fast fourier transform for filtered signal
filtered_signal=np.fft.ifft(frequency_domain_signal_filtered, 1024)
filtered_signal_ready=np.fft.ifft(frequency_domain_signal_filtered_ready, 1024)

#plot(t)
plot(noise_signal)
#plot(frequency_domain_signal)
#plot(G)
#plot(frequency_domain_signal_filtered)
#plot(frequency_domain_signal_filtered_ready)
plot(filtered_signal)
plot(filtered_signal_ready)

show()
```