Tuto_2.1_Matched_filtering_introduction

November 19, 2020

1 Gravitational Wave Open Data Workshop #3

1.1 Tutorial 2.1 PyCBC Tutorial, An introduction to matched-filtering

We will be using the PyCBC library, which is used to study gravitational-wave data, find astrophysical sources due to compact binary mergers, and study their parameters. These are some of the same tools that the LIGO and Virgo collaborations use to find gravitational waves in LIGO/Virgo data

In this tutorial we will walk through how find a specific signal in LIGO data. We present matched filtering as a cross-correlation, in both the time domain and the frequency domain. In the next tutorial (2.2), we use the method as encoded in PyCBC, which is optimal in the case of Gaussian noise and a known signal model. In reality our noise is not entirely Gaussian, and in practice we use a variety of techniques to separate signals from noise in addition to the use of the matched filter.

Click this link to view this tutorial in Google Colaboratory

Additional examples and module level documentation are here

1.2 Installation (un-comment and execute only if running on a cloud platform!)

[1]:
```
# -- Use the following for Google Colab
#! pip install -q 'lalsuite==6.66' 'PyCBC==1.15.3'
```

Important: With Google Colab, you may need to restart the runtime after running the cell above.

1.2.1 Matched-filtering: Finding well modelled signals in Gaussian noise

Matched filtering can be shown to be the optimal method for “detecting” signals—when the signal waveform is known—in Gaussian noise. We’ll explore those assumptions a little later, but for now let’s demonstrate how this works.

Let’s assume you have a stretch of noise, white noise to start:

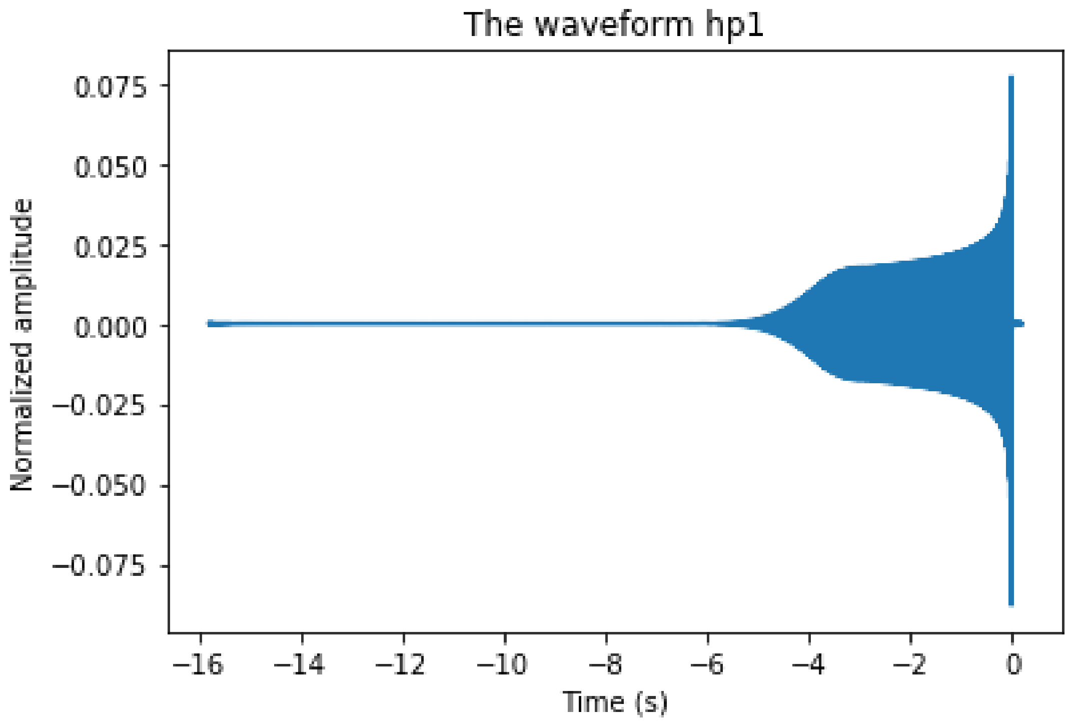
[162]:
```
%matplotlib inline
import numpy
import pylab
```

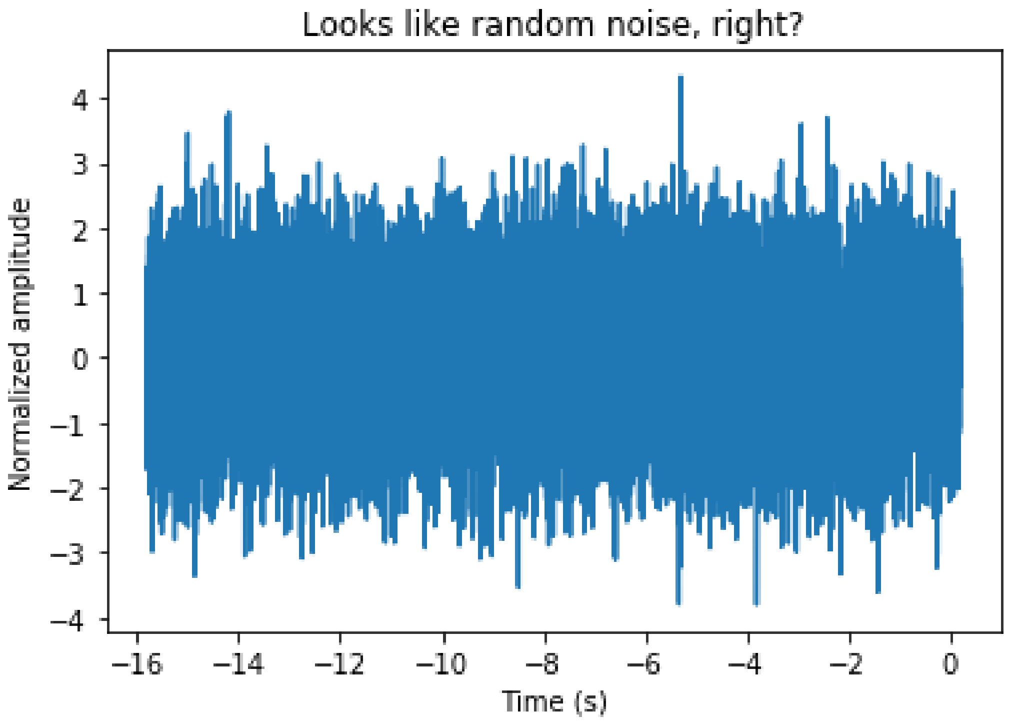
```
# specify the sample rate.
# LIGO raw data is sampled at 16384 Hz (=2^14 samples/second).
# It captures signal frequency content up to f_Nyquist = 8192 Hz.
# Here, we will make the computation faster by sampling at a lower rate.
sample_rate = 1024 # samples per second
data_length = 1024 # seconds

# Generate a long stretch of white noise: the data series and the time series.
data = numpy.random.normal(size=[sample_rate * data_length])
times = numpy.arange(len(data)) / float(sample_rate)
```

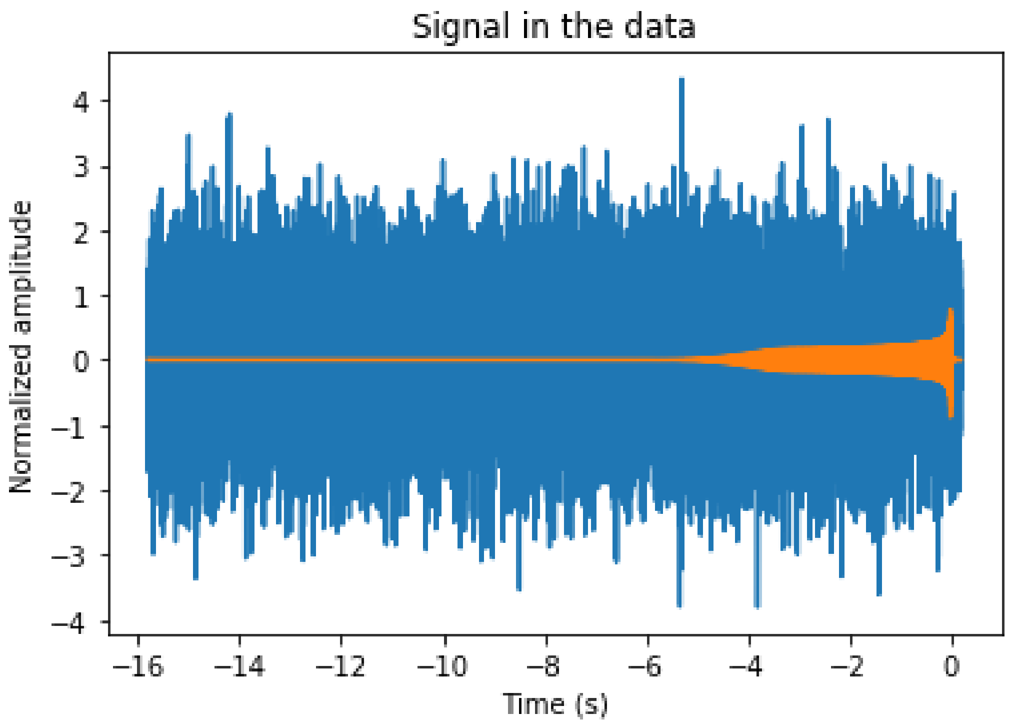
And then let's add a gravitational wave signal to some random part of this data.

[163]:
```
from pycbc.waveform import get_td_waveform

# the "approximant" (jargon for parameterized waveform family).
# IMRPhenomD is defined in the frequency domain, but we'll get it in the time␣
 ↪domain (td).
# It runs fast, but it doesn't include effects such as non-aligned component␣
 ↪spin, or higher order modes.
apx = 'IMRPhenomD'

# You can specify many parameters,
# https://pycbc.org/pycbc/latest/html/pycbc.waveform.html?
 ↪highlight=get_td_waveform#pycbc.waveform.waveform.get_td_waveform
# but here, we'll use defaults for everything except the masses.
# It returns both hplus and hcross, but we'll only use hplus for now.
hp1, _ = get_td_waveform(approximant=apx,
                         mass1=10,
                         mass2=10,
                         delta_t=1.0/sample_rate,
                         f_lower=25)

# The amplitude of gravitational-wave signals is normally of order 1E-20. To␣
 ↪demonstrate our method
# on white noise with amplitude O(1) we normalize our signal so the␣
 ↪cross-correlation of the signal with
# itself will give a value of 1. In this case we can interpret the␣
 ↪cross-correlation of the signal with white
# noise as a signal-to-noise ratio.

hp1 = hp1 / max(numpy.correlate(hp1,hp1, mode='full'))**0.5

# note that in this figure, the waveform amplitude is of order 1.
# The duration (for frequency above f_lower=25 Hz) is only 3 or 4 seconds long.
# The waveform is "tapered": slowly ramped up from zero to full strength, over␣
↪the first second or so.
# It is zero-padded at earlier times.
pylab.figure()
pylab.title("The waveform hp1")
pylab.plot(hp1.sample_times, hp1)
pylab.xlabel('Time (s)')
pylab.ylabel('Normalized amplitude')

# Shift the waveform to start at a random time in the Gaussian noise data.
waveform_start = numpy.random.randint(0, len(data) - len(hp1))
data[waveform_start:waveform_start+len(hp1)] += 10 * hp1.numpy()

pylab.figure()
pylab.title("Looks like random noise, right?")
pylab.plot(hp1.sample_times, data[waveform_start:waveform_start+len(hp1)])
pylab.xlabel('Time (s)')
pylab.ylabel('Normalized amplitude')

pylab.figure()
pylab.title("Signal in the data")
pylab.plot(hp1.sample_times, data[waveform_start:waveform_start+len(hp1)])
pylab.plot(hp1.sample_times, 10 * hp1)
pylab.xlabel('Time (s)')
pylab.ylabel('Normalized amplitude')
```

```
[163]: Text(0, 0.5, 'Normalized amplitude')
```


The waveform hp1
Normalized amplitude
0.075
0.050
0.025
0.000
−0.025
−0.050
−0.075
−16
−14
−12
−10
−8
−6
−4
−2
0
Time (s)


Looks like random noise, right?
Normalized amplitude
4
3
2
1
0
−1
−2
−3
−4
−16
−14
−12
−10
−8
−6
−4
−2
0
Time (s)

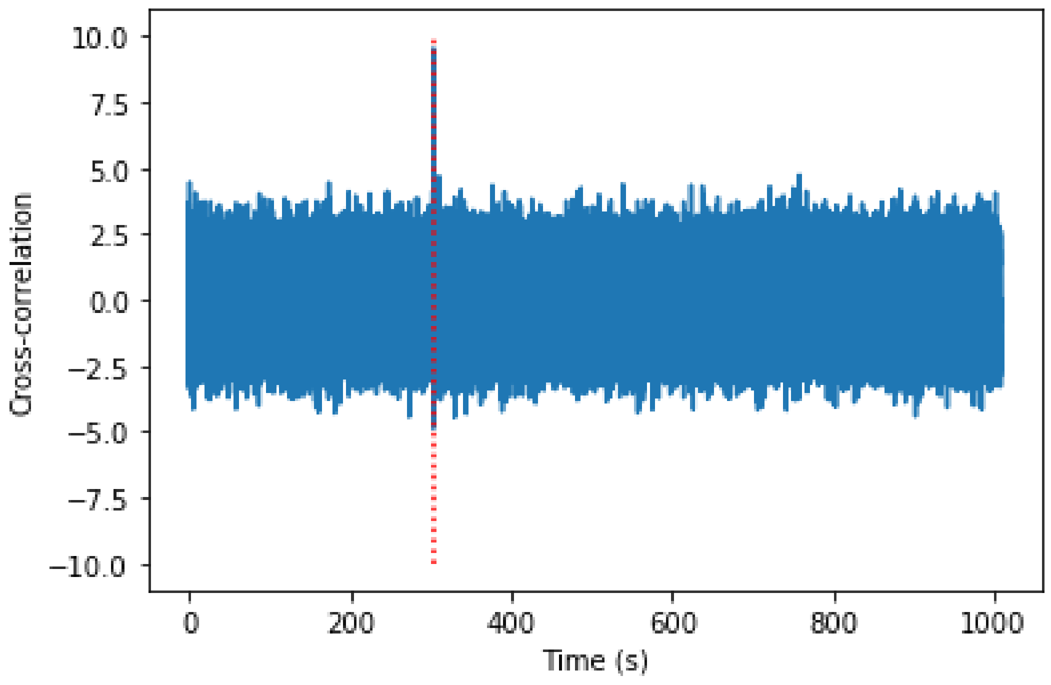


To search for this signal we can cross-correlate the signal with the entire dataset -> Not in any way optimized at this point, just showing the method.

We will do the cross-correlation in the time domain, once for each time step. It runs slowly...

```
[164]: cross_correlation = numpy.zeros([len(data)-len(hp1)])
       hp1_numpy = hp1.numpy()
       for i in range(len(data) - len(hp1_numpy)):
           cross_correlation[i] = (hp1_numpy * data[i:i+len(hp1_numpy)]).sum()

       # plot the cross-correlated data vs time. Superimpose the location of the end␣
        ↪of the signal;
       # this is where we should find a peak in the cross-correlation.
       pylab.figure()
       times = numpy.arange(len(data) - len(hp1_numpy)) / float(sample_rate)
       pylab.plot(times, cross_correlation)
       pylab.plot([waveform_start/float(sample_rate), waveform_start/
        ↪float(sample_rate)], [-10,10],'r:')
       pylab.xlabel('Time (s)')
       pylab.ylabel('Cross-correlation')
```

```
[164]: Text(0, 0.5, 'Cross-correlation')
```

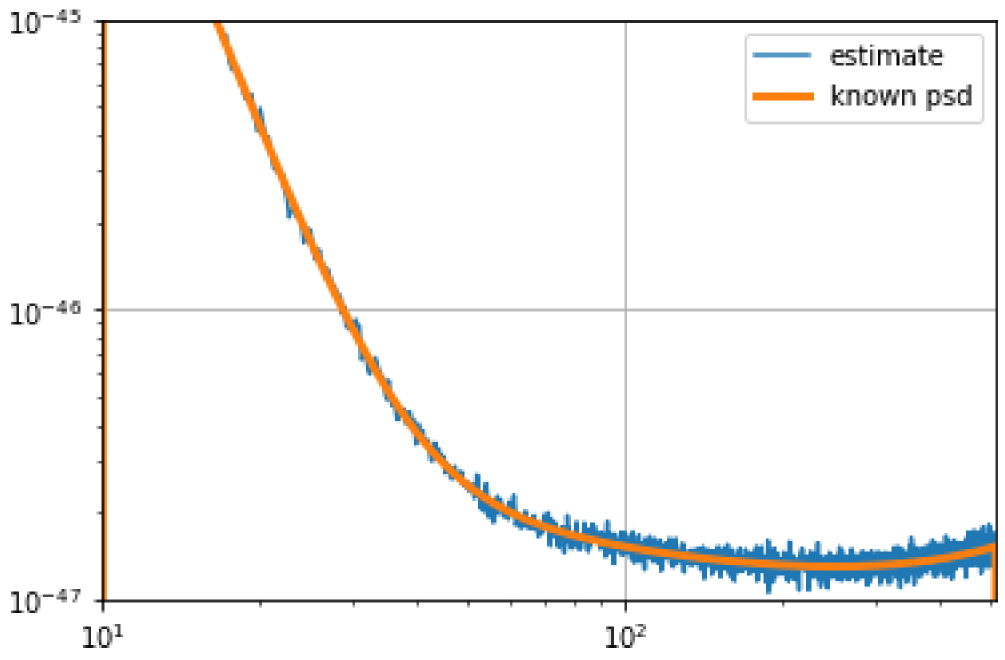


Here you can see that the largest spike from the cross-correlation comes at the time of the signal. We only really need one more ingredient to describe matched-filtering: "Colored" noise (Gaussian noise but with a frequency-dependent variance; white noise has frequency-independent variance).

Let's repeat the process, but generate a stretch of data colored with LIGO's zero-detuned–high-power noise curve. We'll use a PyCBC library to do this.

[165]:
```
# http://pycbc.org/pycbc/latest/html/noise.html
import pycbc.noise
import pycbc.psd

# The color of the noise matches a PSD which you provide:
# Generate a PSD matching Advanced LIGO's zero-detuned--high-power noise curve
flow = 10.0
delta_f = 1.0 / 128
flen = int(sample_rate / (2 * delta_f)) + 1
psd = pycbc.psd.aLIGOZeroDetHighPower(flen, delta_f, flow)

# Generate colored noise
delta_t = 1.0 / sample_rate
ts = pycbc.noise.noise_from_psd(data_length*sample_rate, delta_t, psd, seed=127)

# Estimate the amplitude spectral density (ASD = sqrt(PSD)) for the noisy data
# using the "welch" method. We'll choose 4 seconds PSD samples that are␣
↪overlapped 50%
```

```
seg_len = int(4 / delta_t)
seg_stride = int(seg_len / 2)
estimated_psd = pycbc.psd.welch(ts,seg_len=seg_len,seg_stride=seg_stride)

# plot it:
pylab.loglog(estimated_psd.sample_frequencies, estimated_psd, label='estimate')
pylab.loglog(psd.sample_frequencies, psd, linewidth=3, label='known psd')
pylab.xlim(xmin=flow, xmax=512)
pylab.ylim(1e-47, 1e-45)
pylab.legend()
pylab.grid()
pylab.show()

# add the signal, this time, with a "typical" amplitude.
ts[waveform_start:waveform_start+len(hp1)] += hp1.numpy() * 1E-20
```



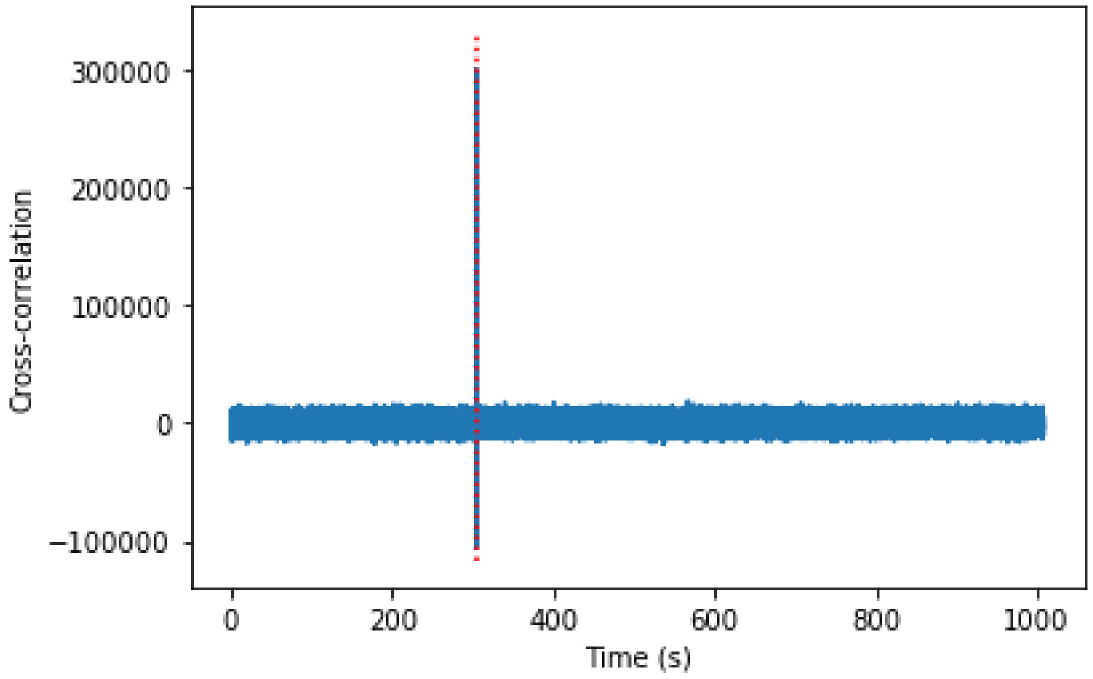
Then all we need to do is to "whiten" both the data, and the template waveform. This can be done, in the frequency domain, by dividing by the PSD. This *can* be done in the time domain as well, but it's more intuitive in the frequency domain

[166]:
```
# Generate a PSD for whitening the data
from pycbc.types import TimeSeries

# The PSD, sampled properly for the noisy data
flow = 10.0
```

```
delta_f = 1.0 / data_length
flen = int(sample_rate / (2 * delta_f)) + 1
psd_td = pycbc.psd.aLIGOZeroDetHighPower(flen, delta_f, 0)

# The PSD, sampled properly for the signal
delta_f = sample_rate / float(len(hp1))
flen = int(sample_rate / (2 * delta_f)) + 1
psd_hp1 = pycbc.psd.aLIGOZeroDetHighPower(flen, delta_f, 0)

# The 0th and Nth values are zero. Set them to a nearby value to avoid dividing␣
 ↪by zero.
psd_td[0] = psd_td[1]
psd_td[len(psd_td) - 1] = psd_td[len(psd_td) - 2]
# Same, for the PSD sampled for the signal
psd_hp1[0] = psd_hp1[1]
psd_hp1[len(psd_hp1) - 1] = psd_hp1[len(psd_hp1) - 2]

# convert both noisy data and the signal to frequency domain,
# and divide each by ASD=PSD**0.5, then convert back to time domain.
# This "whitens" the data and the signal template.
# Multiplying the signal template by 1E-21 puts it into realistic units of␣
 ↪strain.
data_whitened = (ts.to_frequencyseries() / psd_td**0.5).to_timeseries()
hp1_whitened = (hp1.to_frequencyseries() / psd_hp1**0.5).to_timeseries() * 1E-21
```

[167]:
```
# Now let's re-do the correlation, in the time domain, but with whitened data␣
 ↪and template.
cross_correlation = numpy.zeros([len(data)-len(hp1)])
hp1n = hp1_whitened.numpy()
datan = data_whitened.numpy()
for i in range(len(datan) - len(hp1n)):
    cross_correlation[i] = (hp1n * datan[i:i+len(hp1n)]).sum()

# plot the cross-correlation in the time domain. Superimpose the location of␣
 ↪the end of the signal.
# Note how much bigger the cross-correlation peak is, relative to the noise␣
 ↪level,
# compared with the unwhitened version of the same quantity. SNR is much higher!
pylab.figure()
times = numpy.arange(len(datan) - len(hp1n)) / float(sample_rate)
pylab.plot(times, cross_correlation)
pylab.plot([waveform_start/float(sample_rate), waveform_start/
 ↪float(sample_rate)],
           [(min(cross_correlation))*1.1,(max(cross_correlation))*1.1],'r:')
pylab.xlabel('Time (s)')
pylab.ylabel('Cross-correlation')
```

[167]:
```
Text(0, 0.5, 'Cross-correlation')
```



2 Challenge!

- Histogram the whitened time series. Ignoring the outliers associated with the signal, is it a Gaussian? What is the mean and standard deviation? (We have not been careful in normalizing the whitened data properly).
- Histogram the above cross-correlation time series. Ignoring the outliers associated with the signal, is it a Gaussian? What is the mean and standard deviation?
- Find the location of the peak. (Note that here, it can be positive or negative), and the value of the SNR of the signal (which is the absolute value of the peak value, divided by the standard deviation of the cross-correlation time series).

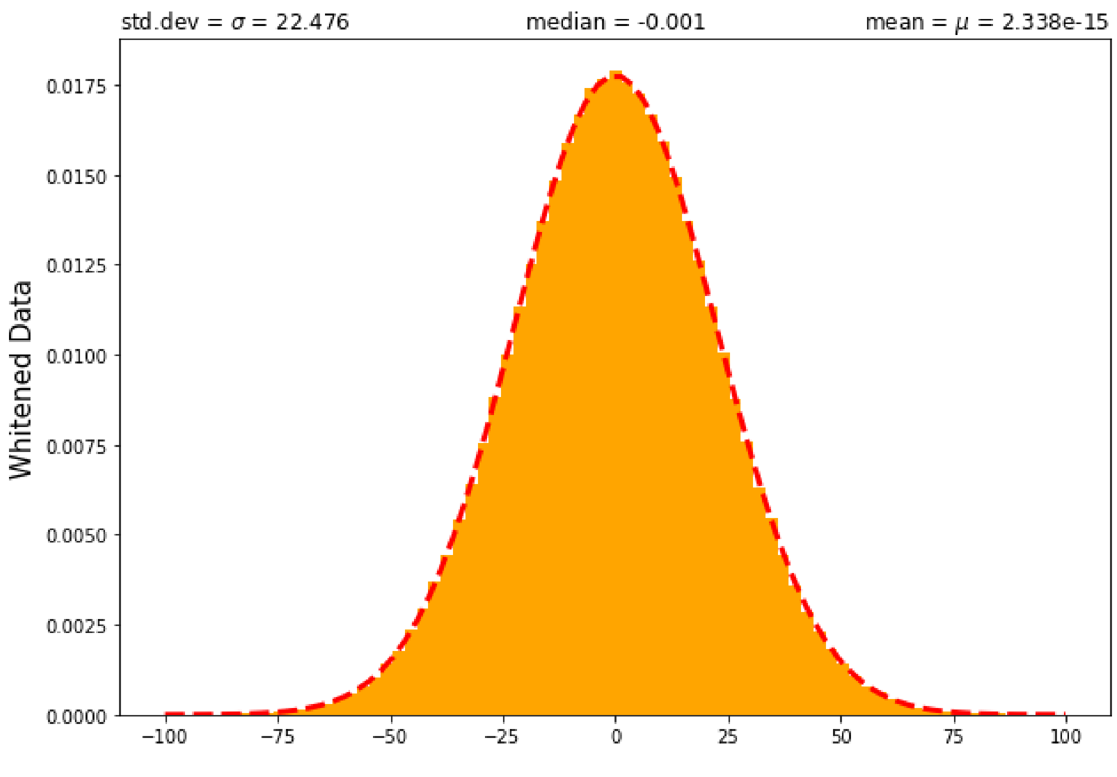
2.1 Optional challenge question. much harder:

- Repeat this process, but instead of using a waveform with mass1=mass2=10, try 15, 20, or 25. Plot the SNR vs mass. Careful! Using lower masses (eg, mass1=mass2=1.4 Msun) will not work here. Why?

[168]:
```python
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.mlab as mlab
from scipy.stats import norm
```

```
fig, ax = plt.subplots(figsize =(10, 7))
n,bins,patches = ax.hist(data_whitened, bins = 75, density=1,␣
↪range=[-100,100],color='orange')
mean = np.mean(data_whitened)
print('mean',mean)
std = np.std(data_whitened)
print('std',std)
median = np.median(data_whitened)
print('median',median)
fit=norm.pdf(bins,mean,std)
ax.plot(bins,fit,'--',color='r', linewidth=3.0)
#ax.set_title(r'$\sigma$ = {}  and  mean = {}' .format(std, mean))
ax.set_title(r'std.dev = $\sigma$ = {0:.3f}' .format(std),loc='left')
ax.set_title(r'mean = $\mu$ = {0:1.3e}' .format(mean),loc='right')
ax.set_title(r'median = {0:1.3f}' .format(median),loc='center')
ax.set_ylabel('Whitened Data',fontsize=15)
plt.show()

#0:1.2f
```

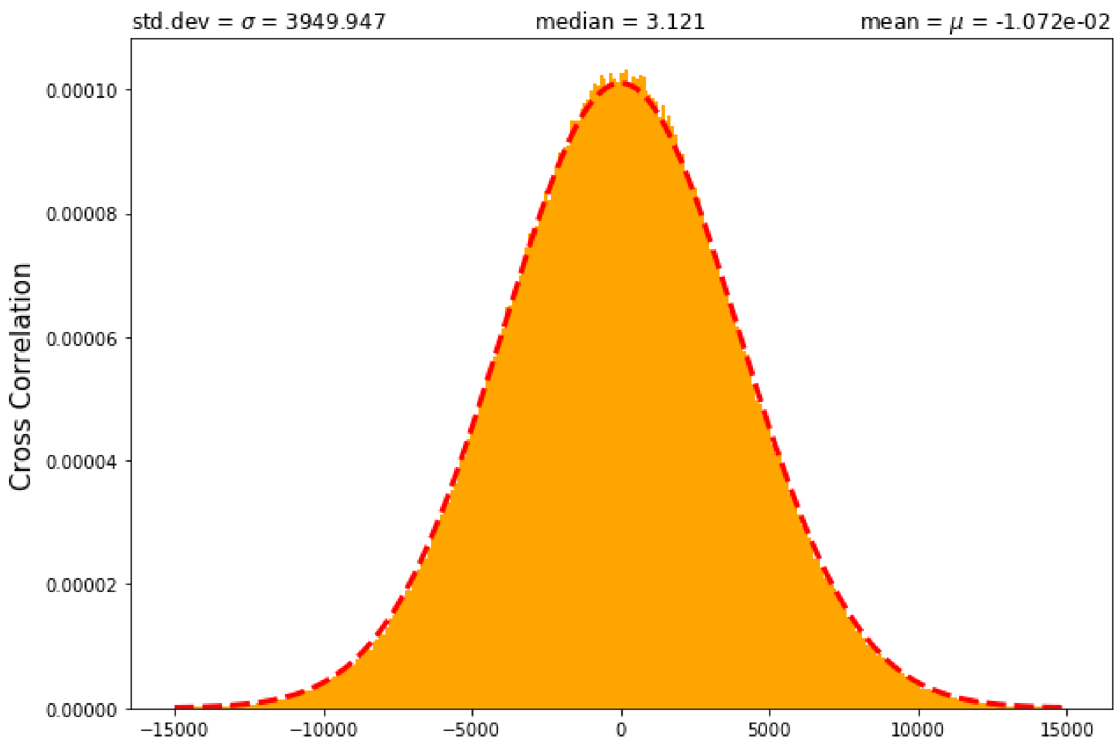
```
mean 2.3375127788244354e-15
std 22.476151014033977
median -0.0014610264932972683
```



Yes the histogram plot of the whitened data is Gaussian

```
[169]: import matplotlib.pyplot as plt
import numpy as np
import matplotlib.mlab as mlab
from scipy.stats import norm
fig, ax = plt.subplots(figsize =(10, 7))
n,bins,patches = ax.hist(cross_correlation, bins = 275, density=1,␣
 ↪range=[-15000,15000],color='orange')
mean = np.mean(cross_correlation)
print('mean',mean)
std = np.std(cross_correlation)
print('std',std)
median = np.median(cross_correlation)
print('median',median)
fit=norm.pdf(bins,mean,std)
ax.plot(bins,fit,'--',color='r', linewidth=3.0)
#ax.set_title(r'$\sigma$ = {}  and  mean = {}' .format(std, mean))
ax.set_title(r'std.dev = $\sigma$ = {0:.3f}' .format(std),loc='left')
ax.set_title(r'mean = $\mu$ = {0:1.3e}' .format(mean),loc='right')
ax.set_title(r'median = {0:1.3f}' .format(median),loc='center')
ax.set_ylabel('Cross Correlation',fontsize=15)
plt.show()
n_max=n.max()
print(n.max())
bin_nmax = np.argmax(n)
SNR_10=n_max/std
print('The SNR_10 value is',SNR_10)
#print(bin_nmax)
```

```
mean -0.010716235257278033
std 3949.9465859950533
median 3.1213582722250965
```



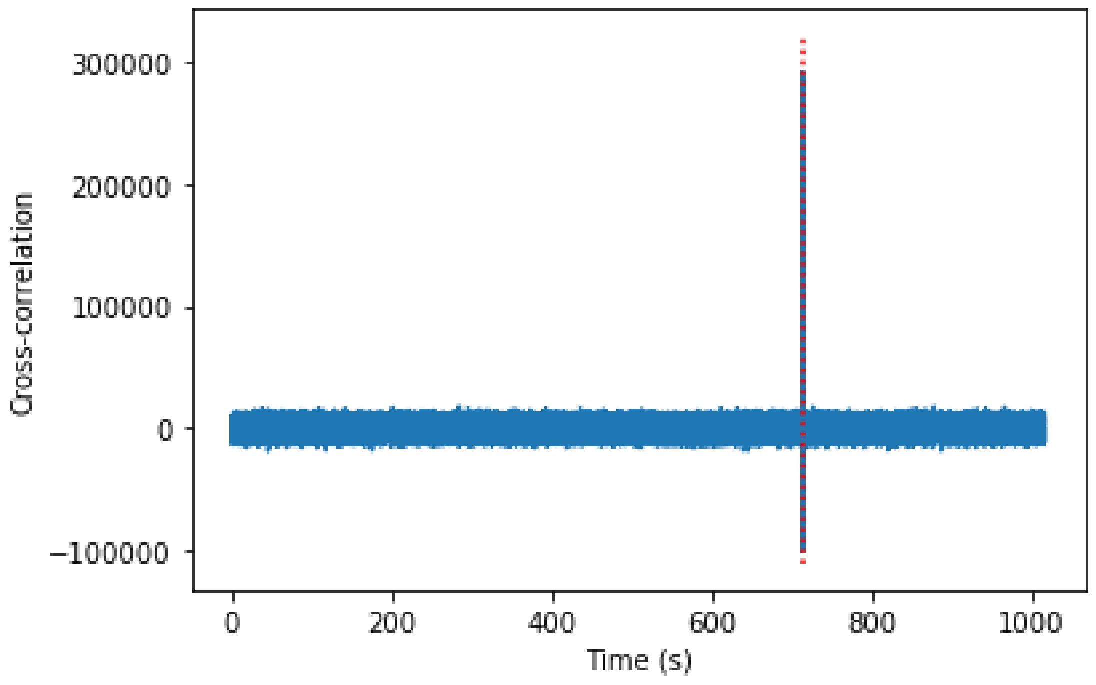
```
0.00010307988867000502
The SNR_10 value is 2.6096527237984812e-08
```

2.1.1 For mass1=mass2=15

[170]:
```
import numpy
import pylab

sample_rate = 1024 # samples per second
data_length = 1024 # seconds

# Generate a long stretch of white noise: the data series and the time series.
data = numpy.random.normal(size=[sample_rate * data_length])
times = numpy.arange(len(data)) / float(sample_rate)

from pycbc.waveform import get_td_waveform

apx = 'IMRPhenomD'

hp1, _ = get_td_waveform(approximant=apx,
                         mass1=15,
                         mass2=15,
                         delta_t=1.0/sample_rate,
                         f_lower=25)

hp1 = hp1 / max(numpy.correlate(hp1,hp1, mode='full'))**0.5


# Shift the waveform to start at a random time in the Gaussian noise data.
waveform_start = numpy.random.randint(0, len(data) - len(hp1))
data[waveform_start:waveform_start+len(hp1)] += 10 * hp1.numpy()

cross_correlation = numpy.zeros([len(data)-len(hp1)])
hp1_numpy = hp1.numpy()
for i in range(len(data) - len(hp1_numpy)):
    cross_correlation[i] = (hp1_numpy * data[i:i+len(hp1_numpy)]).sum()

import pycbc.noise
import pycbc.psd

# The color of the noise matches a PSD which you provide:
# Generate a PSD matching Advanced LIGO's zero-detuned--high-power noise curve
flow = 10.0
delta_f = 1.0 / 128
flen = int(sample_rate / (2 * delta_f)) + 1
psd = pycbc.psd.aLIGOZeroDetHighPower(flen, delta_f, flow)

# Generate colored noise
delta_t = 1.0 / sample_rate
ts = pycbc.noise.noise_from_psd(data_length*sample_rate, delta_t, psd, seed=127)

# Estimate the amplitude spectral density (ASD = sqrt(PSD)) for the noisy data
# using the "welch" method. We'll choose 4 seconds PSD samples that are␣
 ↪overlapped 50%
seg_len = int(4 / delta_t)
seg_stride = int(seg_len / 2)
estimated_psd = pycbc.psd.welch(ts,seg_len=seg_len,seg_stride=seg_stride)




# add the signal, this time, with a "typical" amplitude.
ts[waveform_start:waveform_start+len(hp1)] += hp1.numpy() * 1E-20

# Generate a PSD for whitening the data
from pycbc.types import TimeSeries

# The PSD, sampled properly for the noisy data
flow = 10.0
delta_f = 1.0 / data_length
```

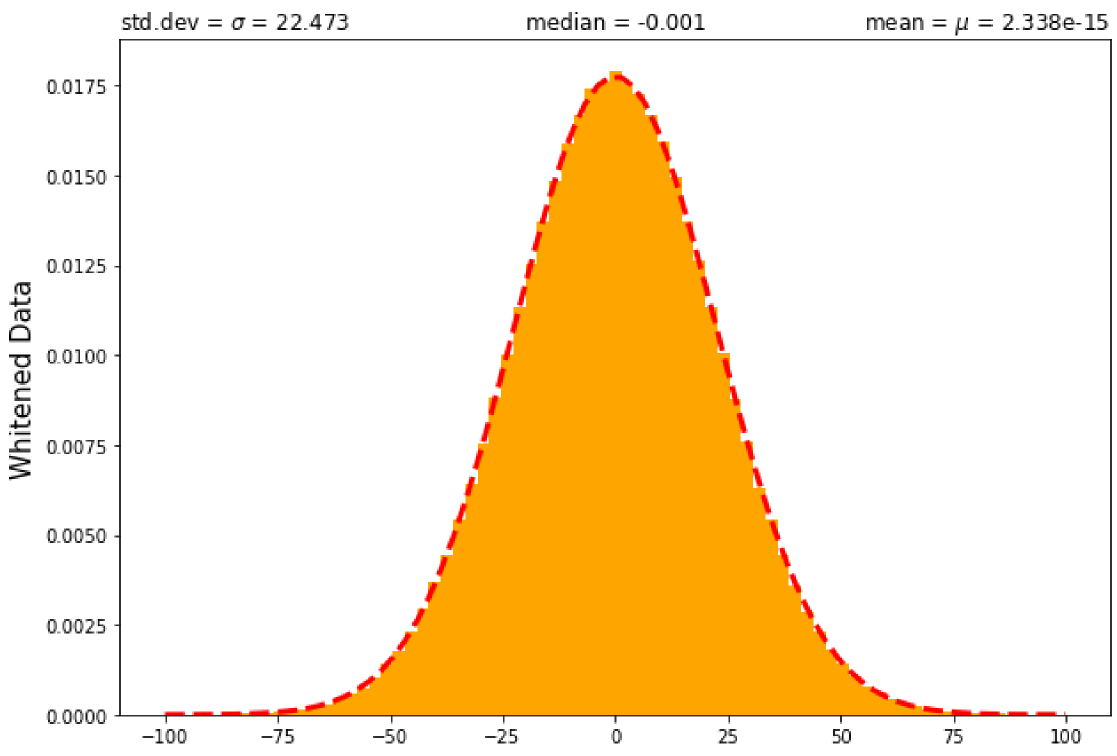
```
flen = int(sample_rate / (2 * delta_f)) + 1
psd_td = pycbc.psd.aLIGOZeroDetHighPower(flen, delta_f, 0)

# The PSD, sampled properly for the signal
delta_f = sample_rate / float(len(hp1))
flen = int(sample_rate / (2 * delta_f)) + 1
psd_hp1 = pycbc.psd.aLIGOZeroDetHighPower(flen, delta_f, 0)

# The 0th and Nth values are zero. Set them to a nearby value to avoid dividing␣
 ↪by zero.
psd_td[0] = psd_td[1]
psd_td[len(psd_td) - 1] = psd_td[len(psd_td) - 2]
# Same, for the PSD sampled for the signal
psd_hp1[0] = psd_hp1[1]
psd_hp1[len(psd_hp1) - 1] = psd_hp1[len(psd_hp1) - 2]

# convert both noisy data and the signal to frequency domain,
# and divide each by ASD=PSD**0.5, then convert back to time domain.
# This "whitens" the data and the signal template.
# Multiplying the signal template by 1E-21 puts it into realistic units of␣
 ↪strain.
data_whitened = (ts.to_frequencyseries() / psd_td**0.5).to_timeseries()
hp1_whitened = (hp1.to_frequencyseries() / psd_hp1**0.5).to_timeseries() * 1E-21

cross_correlation = numpy.zeros([len(data)-len(hp1)])
hp1n = hp1_whitened.numpy()
datan = data_whitened.numpy()
for i in range(len(datan) - len(hp1n)):
    cross_correlation[i] = (hp1n * datan[i:i+len(hp1n)]).sum()

# plot the cross-correlation in the time domain. Superimpose the location of␣
 ↪the end of the signal.
# Note how much bigger the cross-correlation peak is, relative to the noise␣
 ↪level,
# compared with the unwhitened version of the same quantity. SNR is much higher!
pylab.figure()
times = numpy.arange(len(datan) - len(hp1n)) / float(sample_rate)
pylab.plot(times, cross_correlation)
pylab.plot([waveform_start/float(sample_rate), waveform_start/
 ↪float(sample_rate)],
           [(min(cross_correlation))*1.1,(max(cross_correlation))*1.1],'r:')
pylab.xlabel('Time (s)')
pylab.ylabel('Cross-correlation')
```

```
[170]: Text(0, 0.5, 'Cross-correlation')
```



```
[171]: import matplotlib.pyplot as plt
       import numpy as np
       import matplotlib.mlab as mlab
       from scipy.stats import norm
       fig, ax = plt.subplots(figsize =(10, 7))
       n,bins,patches = ax.hist(data_whitened, bins = 75, density=1,␣
        ↪range=[-100,100],color='orange')
       mean = np.mean(data_whitened)
       print('mean',mean)
       std = np.std(data_whitened)
       print('std',std)
       median = np.median(data_whitened)
       print('median',median)
       fit=norm.pdf(bins,mean,std)
       ax.plot(bins,fit,'--',color='r', linewidth=3.0)
       #ax.set_title(r'$\sigma$ = {}  and  mean = {}' .format(std, mean))
       ax.set_title(r'std.dev = $\sigma$ = {0:.3f}' .format(std),loc='left')
       ax.set_title(r'mean = $\mu$ = {0:1.3e}' .format(mean),loc='right')
       ax.set_title(r'median = {0:1.3f}' .format(median),loc='center')
       ax.set_ylabel('Whitened Data',fontsize=15)
       plt.show()
```

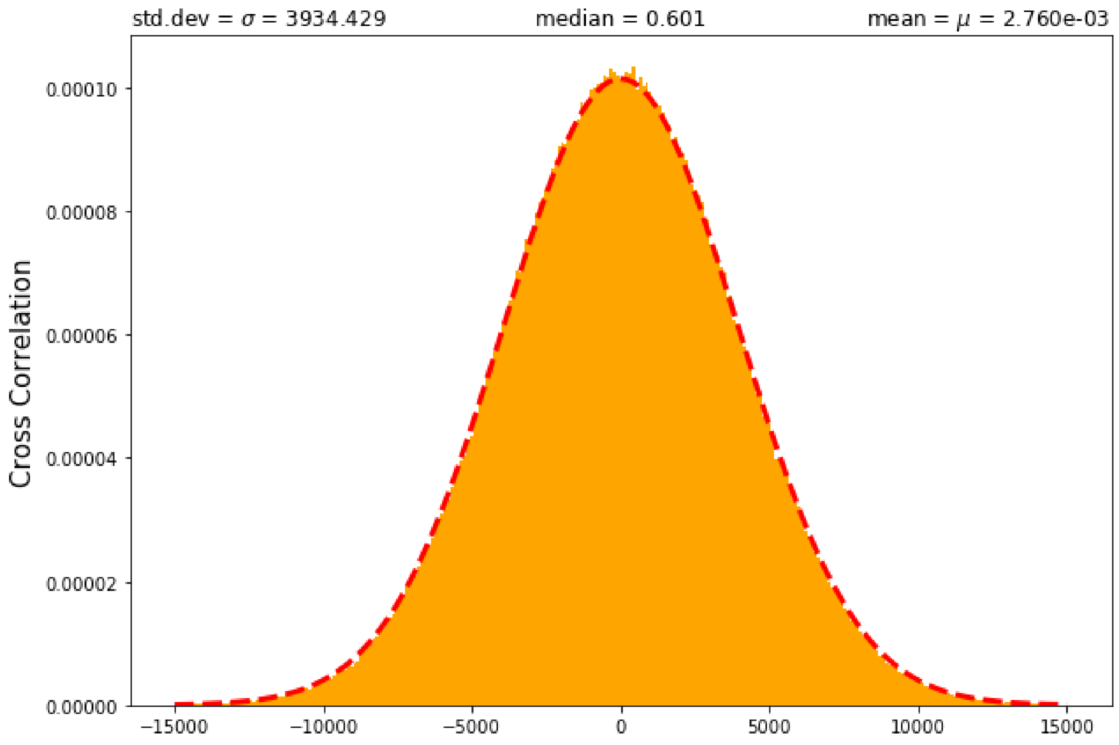
```
mean 2.3379735647477418e-15
std 22.472616925802505
median -0.0013147565196867106
```



[172]:
```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.mlab as mlab
from scipy.stats import norm
fig, ax = plt.subplots(figsize =(10, 7))
n,bins,patches = ax.hist(cross_correlation, bins = 275, density=1,␣
 ↪range=[-15000,15000],color='orange')
mean = np.mean(cross_correlation)
print('mean',mean)
std = np.std(cross_correlation)
print('std',std)
median = np.median(cross_correlation)
print('median',median)
fit=norm.pdf(bins,mean,std)
ax.plot(bins,fit,'--',color='r', linewidth=3.0)
#ax.set_title(r'$\sigma$ = {}  and  mean = {}' .format(std, mean))
ax.set_title(r'std.dev = $\sigma$ = {0:.3f}' .format(std),loc='left')
ax.set_title(r'mean = $\mu$ = {0:1.3e}' .format(mean),loc='right')
ax.set_title(r'median = {0:1.3f}' .format(median),loc='center')
ax.set_ylabel('Cross Correlation',fontsize=15)
plt.show()
n_max=n.max()
print(n.max())
```

```
bin_nmax = np.argmax(n)
SNR_15=n_max/std
print('The SNR_15 value is',SNR_15)
#print(bin_nmax)
```

```
mean 0.0027601267737595524
std 3934.4292473613973
median 0.6012485445767196
```



```
0.00010326339833340305
The SNR_15 value is 2.62460936113303e-08
```

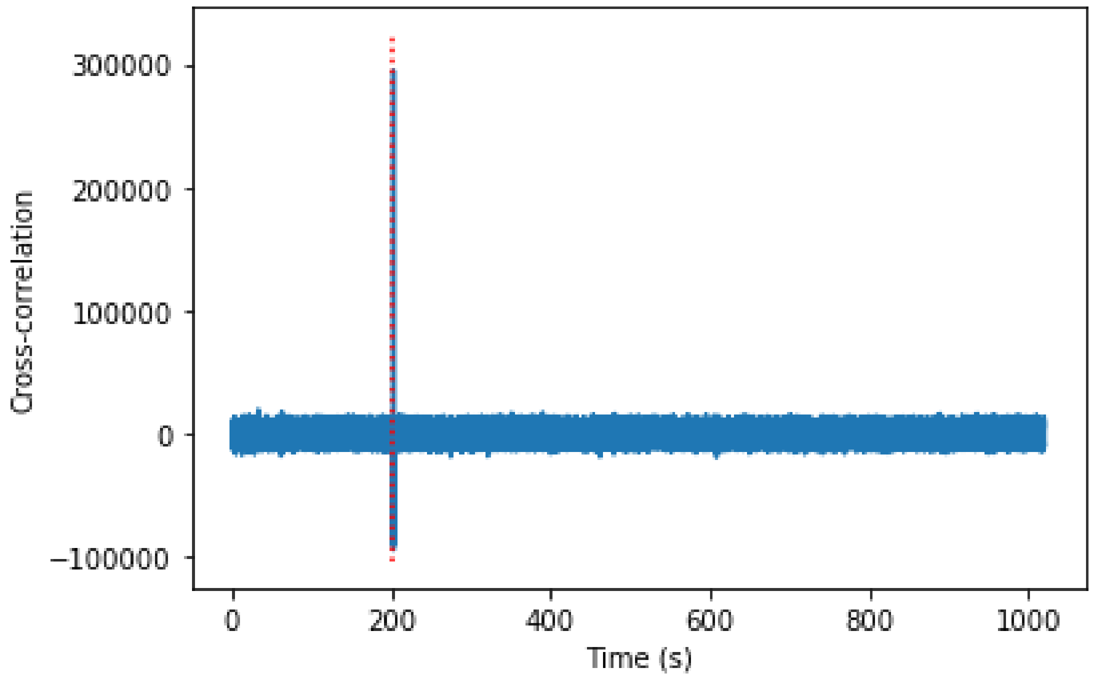
2.1.2 For mass1=mass2=20

[173]:
```
import numpy
import pylab

sample_rate = 1024 # samples per second
data_length = 1024 # seconds

# Generate a long stretch of white noise: the data series and the time series.
data = numpy.random.normal(size=[sample_rate * data_length])
times = numpy.arange(len(data)) / float(sample_rate)
```

```
from pycbc.waveform import get_td_waveform

apx = 'IMRPhenomD'

hp1, _ = get_td_waveform(approximant=apx,
                         mass1=20,
                         mass2=20,
                         delta_t=1.0/sample_rate,
                         f_lower=25)

hp1 = hp1 / max(numpy.correlate(hp1,hp1, mode='full'))**0.5


# Shift the waveform to start at a random time in the Gaussian noise data.
waveform_start = numpy.random.randint(0, len(data) - len(hp1))
data[waveform_start:waveform_start+len(hp1)] += 10 * hp1.numpy()

cross_correlation = numpy.zeros([len(data)-len(hp1)])
hp1_numpy = hp1.numpy()
for i in range(len(data) - len(hp1_numpy)):
    cross_correlation[i] = (hp1_numpy * data[i:i+len(hp1_numpy)]).sum()

import pycbc.noise
import pycbc.psd

# The color of the noise matches a PSD which you provide:
# Generate a PSD matching Advanced LIGO's zero-detuned--high-power noise curve
flow = 10.0
delta_f = 1.0 / 128
flen = int(sample_rate / (2 * delta_f)) + 1
psd = pycbc.psd.aLIGOZeroDetHighPower(flen, delta_f, flow)

# Generate colored noise
delta_t = 1.0 / sample_rate
ts = pycbc.noise.noise_from_psd(data_length*sample_rate, delta_t, psd, seed=127)

# Estimate the amplitude spectral density (ASD = sqrt(PSD)) for the noisy data
# using the "welch" method. We'll choose 4 seconds PSD samples that are␣
 ↪overlapped 50%
seg_len = int(4 / delta_t)
seg_stride = int(seg_len / 2)
estimated_psd = pycbc.psd.welch(ts,seg_len=seg_len,seg_stride=seg_stride)



# add the signal, this time, with a "typical" amplitude.
```

```
ts[waveform_start:waveform_start+len(hp1)] += hp1.numpy() * 1E-20

# Generate a PSD for whitening the data
from pycbc.types import TimeSeries

# The PSD, sampled properly for the noisy data
flow = 10.0
delta_f = 1.0 / data_length
flen = int(sample_rate / (2 * delta_f)) + 1
psd_td = pycbc.psd.aLIGOZeroDetHighPower(flen, delta_f, 0)

# The PSD, sampled properly for the signal
delta_f = sample_rate / float(len(hp1))
flen = int(sample_rate / (2 * delta_f)) + 1
psd_hp1 = pycbc.psd.aLIGOZeroDetHighPower(flen, delta_f, 0)

# The 0th and Nth values are zero. Set them to a nearby value to avoid dividing␣
 ↪by zero.
psd_td[0] = psd_td[1]
psd_td[len(psd_td) - 1] = psd_td[len(psd_td) - 2]
# Same, for the PSD sampled for the signal
psd_hp1[0] = psd_hp1[1]
psd_hp1[len(psd_hp1) - 1] = psd_hp1[len(psd_hp1) - 2]

# convert both noisy data and the signal to frequency domain,
# and divide each by ASD=PSD**0.5, then convert back to time domain.
# This "whitens" the data and the signal template.
# Multiplying the signal template by 1E-21 puts it into realistic units of␣
 ↪strain.
data_whitened = (ts.to_frequencyseries() / psd_td**0.5).to_timeseries()
hp1_whitened = (hp1.to_frequencyseries() / psd_hp1**0.5).to_timeseries() * 1E-21

cross_correlation = numpy.zeros([len(data)-len(hp1)])
hp1n = hp1_whitened.numpy()
datan = data_whitened.numpy()
for i in range(len(datan) - len(hp1n)):
    cross_correlation[i] = (hp1n * datan[i:i+len(hp1n)]).sum()

# plot the cross-correlation in the time domain. Superimpose the location of␣
 ↪the end of the signal.
# Note how much bigger the cross-correlation peak is, relative to the noise␣
 ↪level,
# compared with the unwhitened version of the same quantity. SNR is much higher!
pylab.figure()
times = numpy.arange(len(datan) - len(hp1n)) / float(sample_rate)
pylab.plot(times, cross_correlation)
pylab.plot([waveform_start/float(sample_rate), waveform_start/
 ↪float(sample_rate)],
           [(min(cross_correlation))*1.1,(max(cross_correlation))*1.1],'r:')
pylab.xlabel('Time (s)')
pylab.ylabel('Cross-correlation')
```

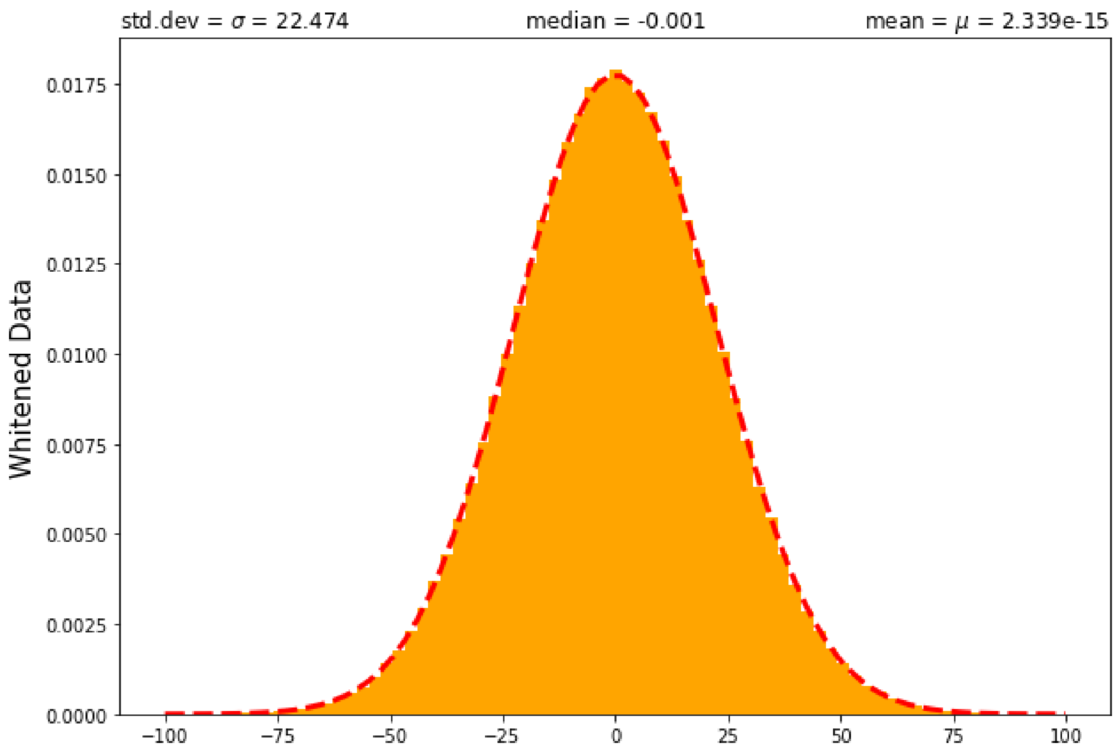
[173]:
```
Text(0, 0.5, 'Cross-correlation')
```



[174]:
```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.mlab as mlab
from scipy.stats import norm
fig, ax = plt.subplots(figsize =(10, 7))
n,bins,patches = ax.hist(data_whitened, bins = 75, density=1,␣
 ↪range=[-100,100],color='orange')
mean = np.mean(data_whitened)
print('mean',mean)
std = np.std(data_whitened)
print('std',std)
median = np.median(data_whitened)
print('median',median)
fit=norm.pdf(bins,mean,std)
ax.plot(bins,fit,'--',color='r', linewidth=3.0)
#ax.set_title(r'$\sigma$ = {}  and  mean = {}' .format(std, mean))
```

```
ax.set_title(r'std.dev = $\sigma$ = {0:.3f}' .format(std),loc='left')
ax.set_title(r'mean = $\mu$ = {0:1.3e}' .format(mean),loc='right')
ax.set_title(r'median = {0:1.3f}' .format(median),loc='center')
ax.set_ylabel('Whitened Data',fontsize=15)
plt.show()
```

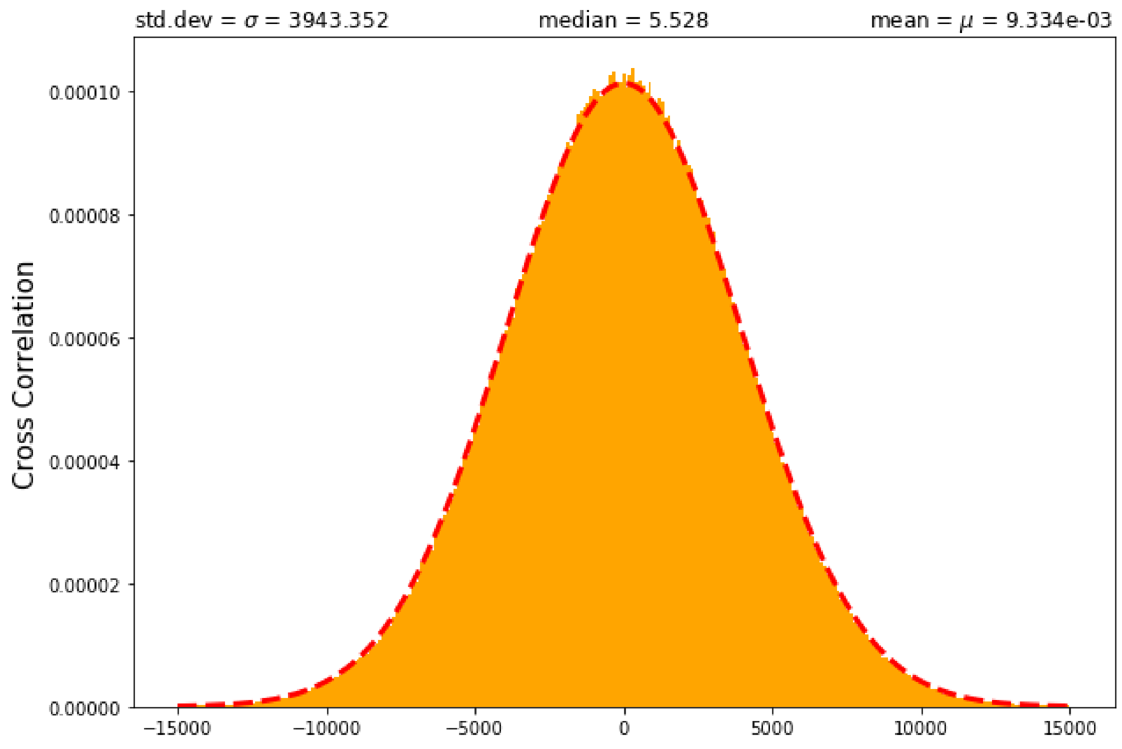
```
mean 2.3387596113227938e-15
std 22.473579875137865
median -0.0009534497979188927
```



[175]:
```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.mlab as mlab
from scipy.stats import norm
fig, ax = plt.subplots(figsize =(10, 7))
n,bins,patches = ax.hist(cross_correlation, bins = 275, density=1,␣
 ↪range=[-15000,15000],color='orange')
mean = np.mean(cross_correlation)
print('mean',mean)
std = np.std(cross_correlation)
print('std',std)
median = np.median(cross_correlation)
print('median',median)
```

```
fit=norm.pdf(bins,mean,std)
ax.plot(bins,fit,'--',color='r', linewidth=3.0)
#ax.set_title(r'$\sigma$ = {}  and  mean = {}' .format(std, mean))
ax.set_title(r'std.dev = $\sigma$ = {0:.3f}' .format(std),loc='left')
ax.set_title(r'mean = $\mu$ = {0:1.3e}' .format(mean),loc='right')
ax.set_title(r'median = {0:1.3f}' .format(median),loc='center')
ax.set_ylabel('Cross Correlation',fontsize=15)
plt.show()
n_max=n.max()
print(n.max())
bin_nmax = np.argmax(n)
SNR_20=n_max/std
print('The SNR_20 value is',SNR_20)
#print(bin_nmax)
```

```
mean 0.009333525984666725
std 3943.351790358998
median 5.5284699273080715
```



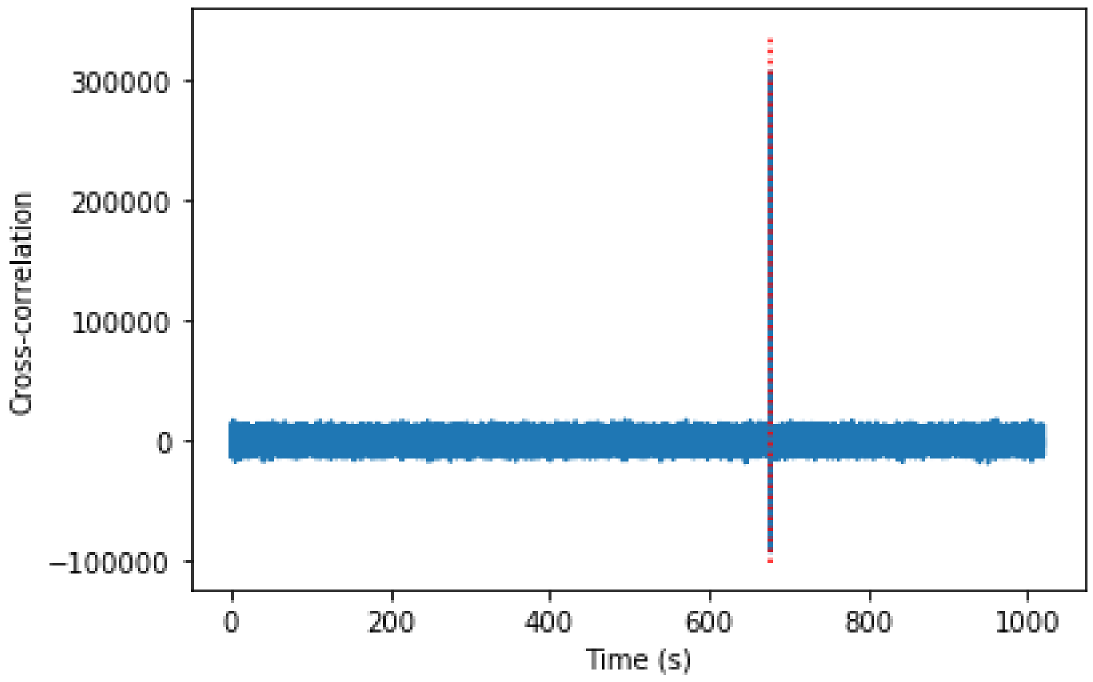
```
0.00010352340579828199
The SNR_20 value is 2.6252642752133796e-08
```

2.1.3 For mass1=mass2=25

[176]:
```
import numpy
import pylab

sample_rate = 1024 # samples per second
data_length = 1024 # seconds

# Generate a long stretch of white noise: the data series and the time series.
data = numpy.random.normal(size=[sample_rate * data_length])
times = numpy.arange(len(data)) / float(sample_rate)

from pycbc.waveform import get_td_waveform

apx = 'IMRPhenomD'

hp1, _ = get_td_waveform(approximant=apx,
                         mass1=25,
                         mass2=25,
                         delta_t=1.0/sample_rate,
                         f_lower=25)

hp1 = hp1 / max(numpy.correlate(hp1,hp1, mode='full'))**0.5


# Shift the waveform to start at a random time in the Gaussian noise data.
waveform_start = numpy.random.randint(0, len(data) - len(hp1))
data[waveform_start:waveform_start+len(hp1)] += 10 * hp1.numpy()

cross_correlation = numpy.zeros([len(data)-len(hp1)])
hp1_numpy = hp1.numpy()
for i in range(len(data) - len(hp1_numpy)):
    cross_correlation[i] = (hp1_numpy * data[i:i+len(hp1_numpy)]).sum()

import pycbc.noise
import pycbc.psd

# The color of the noise matches a PSD which you provide:
# Generate a PSD matching Advanced LIGO's zero-detuned--high-power noise curve
flow = 10.0
delta_f = 1.0 / 128
flen = int(sample_rate / (2 * delta_f)) + 1
psd = pycbc.psd.aLIGOZeroDetHighPower(flen, delta_f, flow)

# Generate colored noise
delta_t = 1.0 / sample_rate
ts = pycbc.noise.noise_from_psd(data_length*sample_rate, delta_t, psd, seed=127)
```

```
# Estimate the amplitude spectral density (ASD = sqrt(PSD)) for the noisy data
# using the "welch" method. We'll choose 4 seconds PSD samples that are␣
 ↪overlapped 50%
seg_len = int(4 / delta_t)
seg_stride = int(seg_len / 2)
estimated_psd = pycbc.psd.welch(ts,seg_len=seg_len,seg_stride=seg_stride)



# add the signal, this time, with a "typical" amplitude.
ts[waveform_start:waveform_start+len(hp1)] += hp1.numpy() * 1E-20

# Generate a PSD for whitening the data
from pycbc.types import TimeSeries

# The PSD, sampled properly for the noisy data
flow = 10.0
delta_f = 1.0 / data_length
flen = int(sample_rate / (2 * delta_f)) + 1
psd_td = pycbc.psd.aLIGOZeroDetHighPower(flen, delta_f, 0)

# The PSD, sampled properly for the signal
delta_f = sample_rate / float(len(hp1))
flen = int(sample_rate / (2 * delta_f)) + 1
psd_hp1 = pycbc.psd.aLIGOZeroDetHighPower(flen, delta_f, 0)

# The 0th and Nth values are zero. Set them to a nearby value to avoid dividing␣
 ↪by zero.
psd_td[0] = psd_td[1]
psd_td[len(psd_td) - 1] = psd_td[len(psd_td) - 2]
# Same, for the PSD sampled for the signal
psd_hp1[0] = psd_hp1[1]
psd_hp1[len(psd_hp1) - 1] = psd_hp1[len(psd_hp1) - 2]

# convert both noisy data and the signal to frequency domain,
# and divide each by ASD=PSD**0.5, then convert back to time domain.
# This "whitens" the data and the signal template.
# Multiplying the signal template by 1E-21 puts it into realistic units of␣
 ↪strain.
data_whitened = (ts.to_frequencyseries() / psd_td**0.5).to_timeseries()
hp1_whitened = (hp1.to_frequencyseries() / psd_hp1**0.5).to_timeseries() * 1E-21

cross_correlation = numpy.zeros([len(data)-len(hp1)])
hp1n = hp1_whitened.numpy()
datan = data_whitened.numpy()
for i in range(len(datan) - len(hp1n)):
    cross_correlation[i] = (hp1n * datan[i:i+len(hp1n)]).sum()

# plot the cross-correlation in the time domain. Superimpose the location of␣
↪the end of the signal.
# Note how much bigger the cross-correlation peak is, relative to the noise␣
↪level,
# compared with the unwhitened version of the same quantity. SNR is much higher!
pylab.figure()
times = numpy.arange(len(datan) - len(hp1n)) / float(sample_rate)
pylab.plot(times, cross_correlation)
pylab.plot([waveform_start/float(sample_rate), waveform_start/
↪float(sample_rate)],
           [(min(cross_correlation))*1.1,(max(cross_correlation))*1.1],'r:')
pylab.xlabel('Time (s)')
pylab.ylabel('Cross-correlation')
```

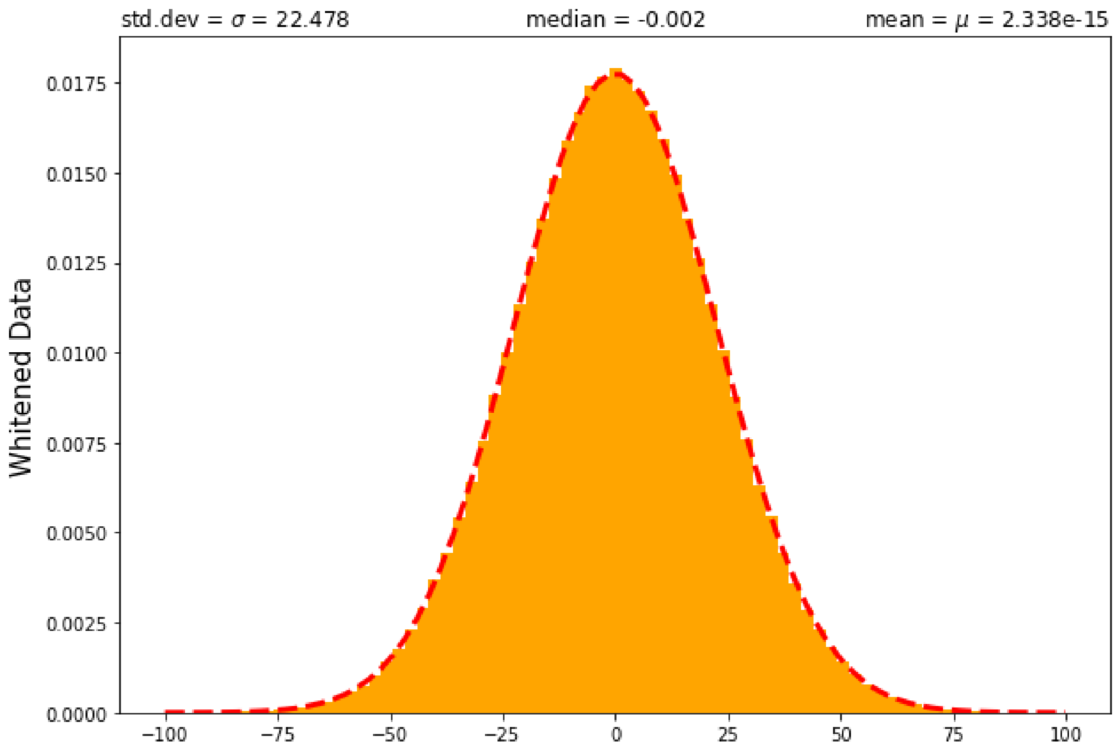
[176]:
```
Text(0, 0.5, 'Cross-correlation')
```



[177]:
```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.mlab as mlab
from scipy.stats import norm
fig, ax = plt.subplots(figsize =(10, 7))
```

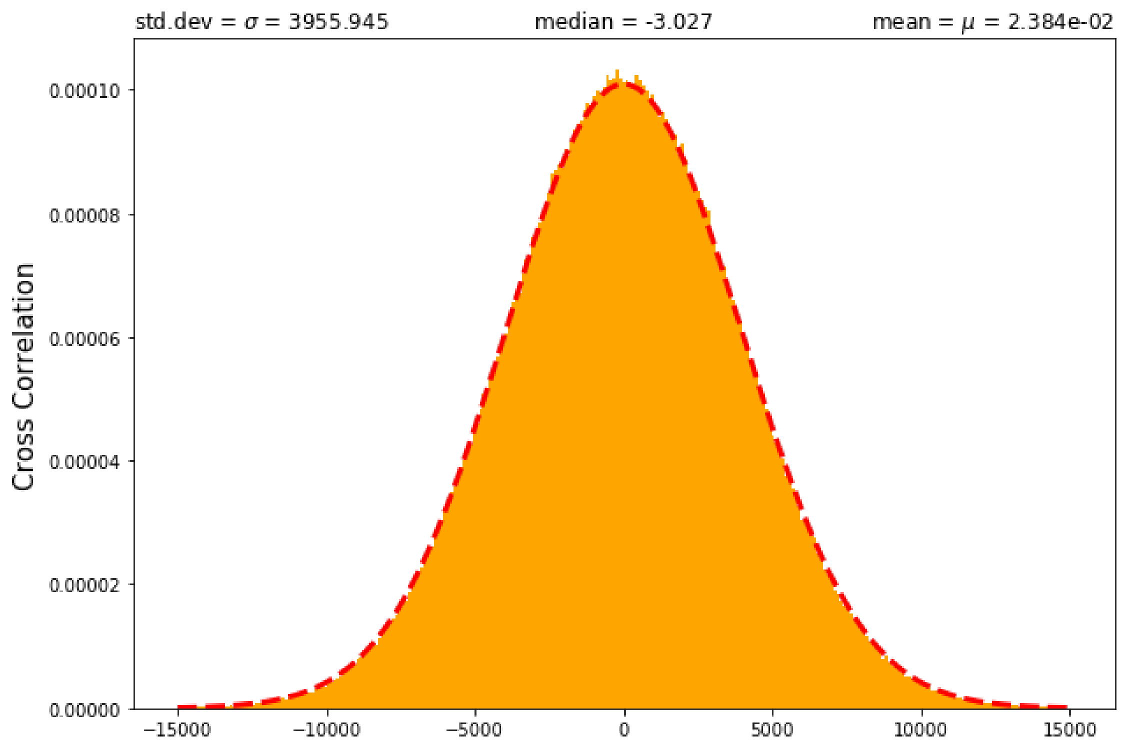
```
n,bins,patches = ax.hist(data_whitened, bins = 75, density=1,␣
 ↪range=[-100,100],color='orange')
mean = np.mean(data_whitened)
print('mean',mean)
std = np.std(data_whitened)
print('std',std)
median = np.median(data_whitened)
print('median',median)
fit=norm.pdf(bins,mean,std)
ax.plot(bins,fit,'--',color='r', linewidth=3.0)
#ax.set_title(r'$\sigma$ = {}  and  mean = {}' .format(std, mean))
ax.set_title(r'std.dev = $\sigma$ = {0:.3f}' .format(std),loc='left')
ax.set_title(r'mean = $\mu$ = {0:1.3e}' .format(mean),loc='right')
ax.set_title(r'median = {0:1.3f}' .format(median),loc='center')
ax.set_ylabel('Whitened Data',fontsize=15)
plt.show()
```

```
mean 2.3383259304537996e-15
std 22.478357845208603
median -0.0015442773468294924
```



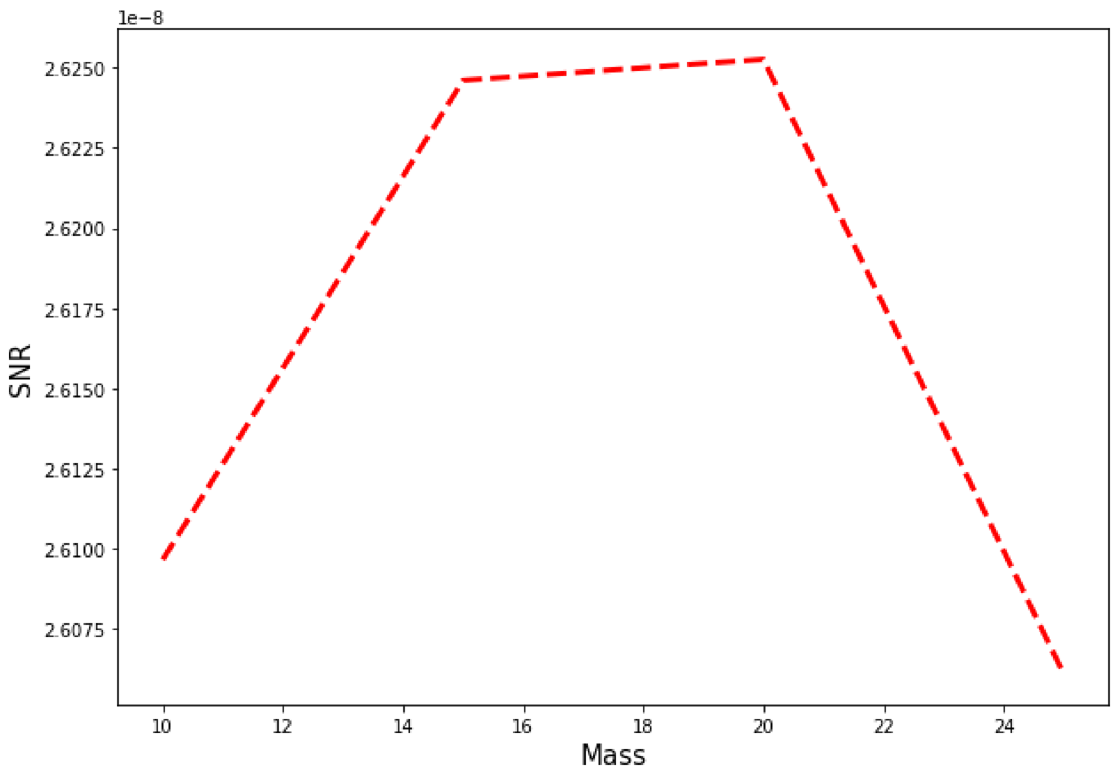
```
[178]: import matplotlib.pyplot as plt
       import numpy as np
       import matplotlib.mlab as mlab
       from scipy.stats import norm
       fig, ax = plt.subplots(figsize =(10, 7))
       n,bins,patches = ax.hist(cross_correlation, bins = 275, density=1,␣
        ↪range=[-15000,15000],color='orange')
       mean = np.mean(cross_correlation)
       print('mean',mean)
       std = np.std(cross_correlation)
       print('std',std)
       median = np.median(cross_correlation)
       print('median',median)
       fit=norm.pdf(bins,mean,std)
       ax.plot(bins,fit,'--',color='r', linewidth=3.0)
       #ax.set_title(r'$\sigma$ = {}  and  mean = {}' .format(std, mean))
       ax.set_title(r'std.dev = $\sigma$ = {0:.3f}' .format(std),loc='left')
       ax.set_title(r'mean = $\mu$ = {0:1.3e}' .format(mean),loc='right')
       ax.set_title(r'median = {0:1.3f}' .format(median),loc='center')
       ax.set_ylabel('Cross Correlation',fontsize=15)
       plt.show()
       n_max=n.max()
       print(n.max())
       bin_nmax = np.argmax(n)
       SNR_25=n_max/std
       print('The SNR_25 value is',SNR_25)
       #print(bin_nmax)
```

```
mean 0.023842330973104828
std 3955.94497227049
median -3.0268976828298197
```



```
0.00010309567105909619
The SNR_25 value is 2.6060946697123816e-08
```

[184]:
```
SNR=[SNR_10,SNR_15,SNR_20,SNR_25]
Mass=[10,15,20,25]
fig, ax = plt.subplots(figsize =(10, 7))
ax.plot(Mass,SNR,'--',color='r', linewidth=3.0)
#ax.set_title(r'$\sigma$ = {}  and  mean = {}' .format(std, mean))
ax.set_xlabel('Mass',fontsize=15)
ax.set_ylabel('SNR',fontsize=15)
#ax.set_yscale('log')
#ax.set_xscale('log')

plt.show()
```



2.1.4 Optimizing a matched-filter

That's all that a matched-filter is. A cross-correlation of the data with a template waveform performed as a function of time. This cross-correlation walking through the data is a convolution operation. Convolution operations are more optimally performed in the frequency domain, which becomes a `O(N ln N)` operation, as opposed to the `O(N^2)` operation shown here. You can also conveniently vary the phase of the signal in the frequency domain, as we will illustrate in the next tutorial. PyCBC implements a frequency-domain matched-filtering engine, which is much faster than the code we've shown here. Let's move to the next tutorial now, where we will demonstrate its use on real data.