Problem set 6

In [115…
```
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.simplefilter("ignore")
```

Question 1

To shift the function, we use the property of the Dirac delta function:

$$f(y) = \int f(x)\delta(y-x)dx \tag{1}$$

together with the definition of the convolution seen in the assignment. We see that $g(y-x) = \delta(y-x)$, and that since the DFT of δ is a plane wave, the product to inverse Fourier transform will simply be

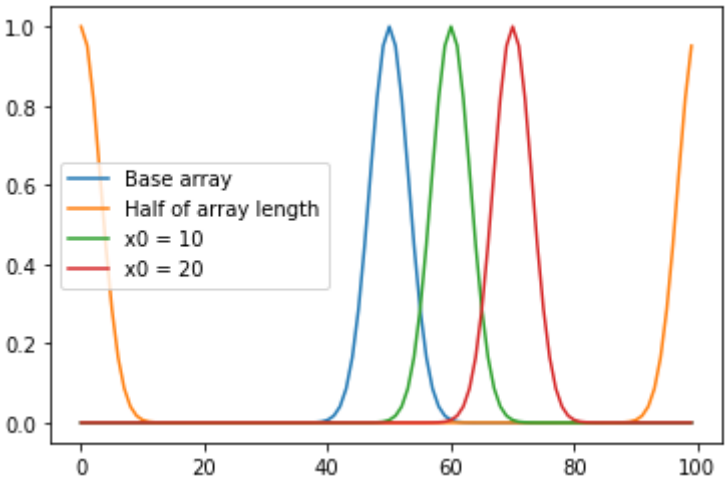
$$F(k) * \exp\left(2\pi i x_0 k/N\right). \tag{2}$$

In [274…
```
def shift_conv(a,x0):
    """
    shifts an array `a` by a length x0
    `a` is 1-d
    assume a is real
    """
    # set the number of points
    N = np.size(a) # `a` is 1-d

    # dft of a
    aft = np.fft.fft(a)

    # dft of delta
    ks = np.arange(N)
    plane = np.exp(-2*np.pi*1j*x0*ks/N) # - sign is crucial to get correct shi

    # idft of their product is the shifted array by definition
    shift_a = np.real(np.fft.ifft(aft*plane))

    return shift_a
```

Let's plot this to check it

In [278…
```
# set no. of points
N = 100
xs = np.arange(N)

# gaussian centred at half
gauss = np.exp(-0.5*(xs-N//2)**2/10)

# shift is half of array
```

```
shift = N//2
gauss_shift_half = shift_conv(gauss,shift)
gauss_shift_10 = shift_conv(gauss,10)
gauss_shift_20 = shift_conv(gauss,20)

# plot to check
plt.plot(xs,gauss,label="Base array")
plt.plot(xs,gauss_shift_half,label="Half of array length")
plt.plot(xs,gauss_shift_10,label="x0 = 10")
plt.plot(xs,gauss_shift_20,label="x0 = 20")
plt.legend()
plt.show()
# plt.xlim(40,60)
```



The shift of $x_0 =$ half array length is halved on each end due to the wrap-around nature of the DFT.

Question 2

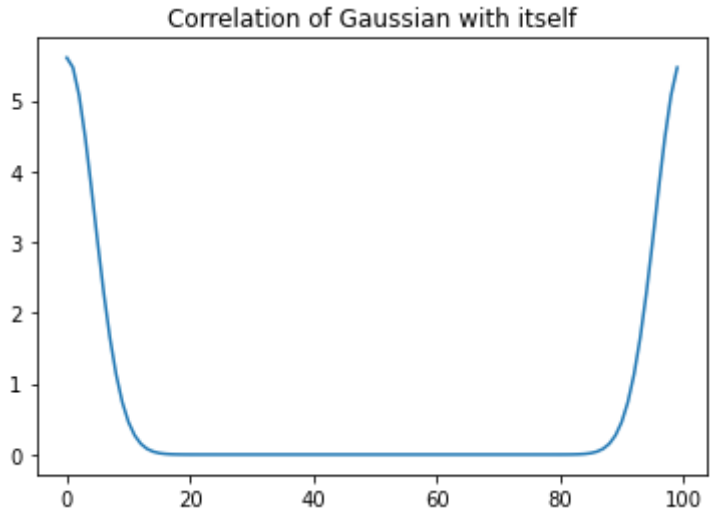
a)

In [280…
```
def correlation_function(a,b):
    """
    computes the correlation function of two arrays a,b
    """
    aft = np.fft.fft(a)
    bft = np.fft.fft(b)
    corr = np.fft.ifft(aft*np.conj(bft))

    return corr
```

In [283…
```
# gaussian
gauss_one = np.exp(-0.5*(xs-N//2)**2/10) # divide by 10 to add some width

# correlation function with itself
gauss_corr = correlation_function(gauss_one,gauss_one)
```

In [284…
```
plt.title("Correlation of Gaussian with itself")
plt.plot(xs,gauss_corr)
```

Out[284]: [<matplotlib.lines.Line2D at 0x7f9b6cff7d90>]

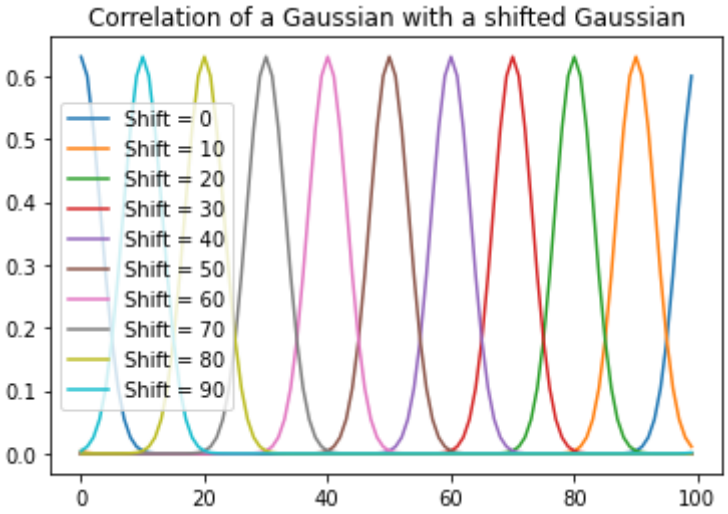


b)

In [285…
```
def correlation_shifted(a,x0):
    """
    routine that takes gaussian and correlates with shifted gaussian
    N is the number of total points
    """
    N = np.size(a)
    xs = np.arange(N)
    arr_shift = shift_conv(a,x0)

    corr = correlation_function(a,arr_shift)

    return corr
```

In [288…
```
N = 100
gauss = np.exp(-0.5*(xs-N//2)**2/5) / np.sqrt(2*np.pi)
```

In [291…
```
plt.title("Correlation of a Gaussian with a shifted Gaussian")
for i in range(10):
    x0 = 10*i
    plt.plot(np.arange(N),correlation_shifted(a=gauss,x0=x0),label="Shift = {}'
plt.legend()
plt.show()
```



The center of the correlation function depends on the relative shift between the two correlated Gaussians. When the shift is zero, the correlation function is centred at 0. When the shift is 50 (brown line) the correlation function is centred at 50, and so on. This is not surprising, since that is where both Gaussians have most of their weight.

Question 3

To avoid any wrapping around, let's do as the problem states and simply add zeros to the end of our input arrays.

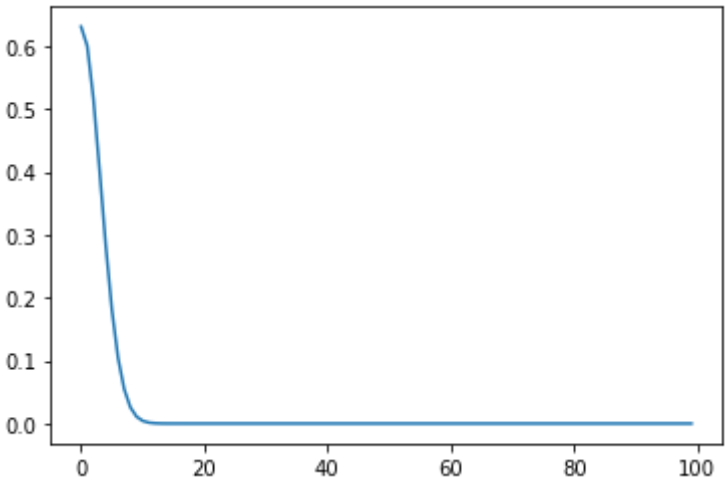
In [292…

```
def array_zeroed(a):
    """
    Returns an array padded with extra zeroes at the end
    """
    N = np.size(a)
    a_zeroed = np.hstack([a,np.zeros(N)])

    return a_zeroed

def convolution(a,b):
    """
    Takes the convolution of two arrays
    """
    aft = np.fft.fft(a)
    bft = np.fft.fft(b)
    conv = np.fft.ifft(aft*bft)

    return conv

def conv_no_wrap(a,b):
    """
    Convolution without any danger of wrapping around
    """
    N = np.size(a)

    a_pad = array_zeroed(a)
    b_pad = array_zeroed(b)
```

```
    conv = convolution(a_pad,b_pad)

    return conv[N:] # adjust for padding
```

Let's test it out with the Gaussian from the previous problem:

In [294…
```
plt.plot(xs,conv_no_wrap(gauss,shift_conv(gauss,0)))
plt.show()
```



Indeed, there is no wrapping!

Question 4

a)

For this, we can use the geomtric sum formula $\sum_{x=0}^{n} r^x = \frac{1-r^{n+1}}{1-r}$. In our case, r is simply $r = \exp\left(-2\pi ik/N\right)$. Therefore,

$$\sum_{x=0}^{N-1} \exp\left(-2\pi ikx/N\right) = \frac{1-\exp\left(-2\pi ik\right)}{1-\exp\left(-2\pi ik/N\right)} \tag{3}$$

as desired.

b)

Obviously, plugging in $k=0$ gives us $0/0$. So, we need to use L'Hospitals rule and differentiate both the numerator and denominator of the expression. For the numerator, the derivative is $-2\pi i \exp\left(-2\pi ik\right)$. The denominator is $-2\pi i/N \exp\left(-2\pi ik/N\right)$. The ratio is therefore:

$$\lim_{k\to 0} \frac{1-\exp\left(-2\pi ik\right)}{1-\exp\left(-2\pi ik/N\right)} = \lim_{k\to 0} \frac{-2\pi i \exp\left(-2\pi ik\right)}{-2\pi i/N \exp\left(-2\pi ik/N\right)} = N \tag{4}$$

as desired.

If k is a multiple of N, the exponential in the sum will take the form of $\exp 2\pi inx$ where n is an integer. Therefore, it is simply a sum of 1s, adding up to N. If k is *not* a multiple of N however, the numerator $1-\exp -2\pi ik$ is still 0 for an integer k, whereas the denominator is not zero because k/N is not an integer. Therefore, we have $0/a$ where a is some complex number, and this is obviously 0. Alternatively, the terms in the complex sums will all cancel each other out since they are phasors in the complex unit circle which are the Nth root of unity, and they are summed N times: this is identically zero! To illustrate, think of the four sums over the fourth root of unity $\sum_{x=0}^{3} \exp \pi ix/2 = 1+i-1-i=0$.

c)

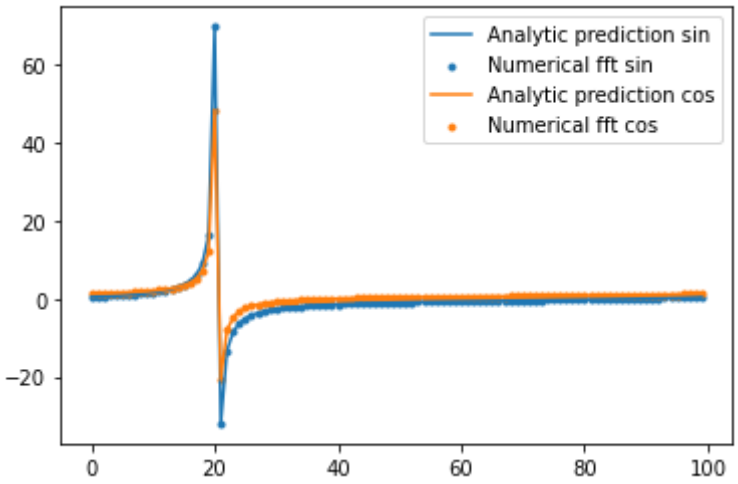
Let's pick $k=1/2$. The function to DFT is therefore $\sin 2\pi kx/N = \sin \pi x/N$. We can use a trick and take the function to be $\exp ipix/N$ and finally take the imaginary part, since the even DFT (i.e. with the cosine) is simply zero. Therefore, we have

$$F(k) = \Im \sum_{x=0}^{N-1} \exp\left(-2\pi ikx/N\right) \exp\left(\pi ix/N\right) = \Im \sum_{x=0}^{N-1} \exp\left(-2\pi i(k-1/2)x/N\right) = \Im$$

Let's plot this as a function of k:

In [295…
```
N = 100
ks = np.arange(N)
xs = np.arange(N)
k0 = 20.31 # why not
ratio = (1-np.exp(-2*np.pi*1j*(ks-k0)))/(1-np.exp(-2*np.pi*1j*(ks-k0)/N))
dft_sin = np.imag(ratio)
dft_cos = np.real(ratio)

# complex sine wave
etothe = np.exp(2*np.pi*1j*k0*xs/N)

# numerical DFTs
num_sin = np.imag(np.fft.fft(etothe))
num_cos = np.real(np.fft.fft(etothe))
```

In [296…
```
plt.plot(ks,dft_sin,label="Analytic prediction sin")
plt.scatter(ks,num_sin,label="Numerical fft sin",marker='.')

plt.plot(ks,dft_cos,label="Analytic prediction cos")
plt.scatter(ks,num_cos,label="Numerical fft cos",marker='.')


plt.legend()
plt.show()
```

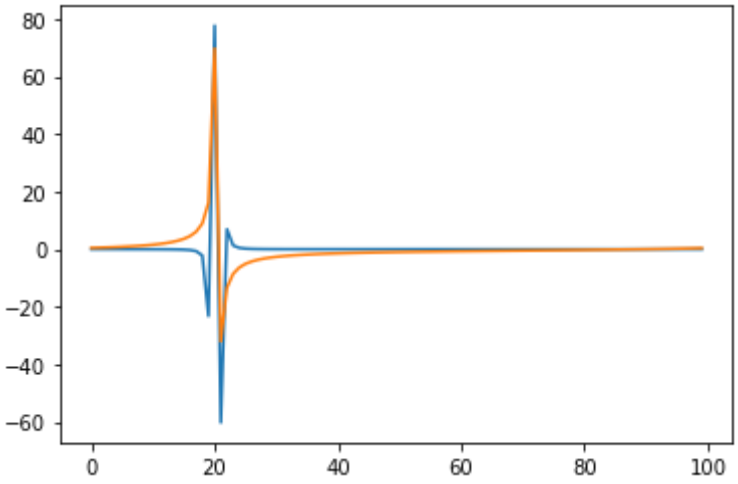


We see that this is not exactly a dirac delta like we would expect for integer values of k. we are nontetheless close to this result, though there is a finite width which we call "spectral leakage".

d)

In [297…
```
window = 0.5 - 0.5*np.cos(2*np.pi*xs/N)
window = window / np.mean(window)

sin_adjusted = window*etothe

num_sin_adj = np.imag(np.fft.fft(sin_adjusted))
```

In [298…
```
plt.plot(ks,num_sin_adj)
plt.plot(ks,num_sin,label="Numerical fft sin")
plt.show()
```



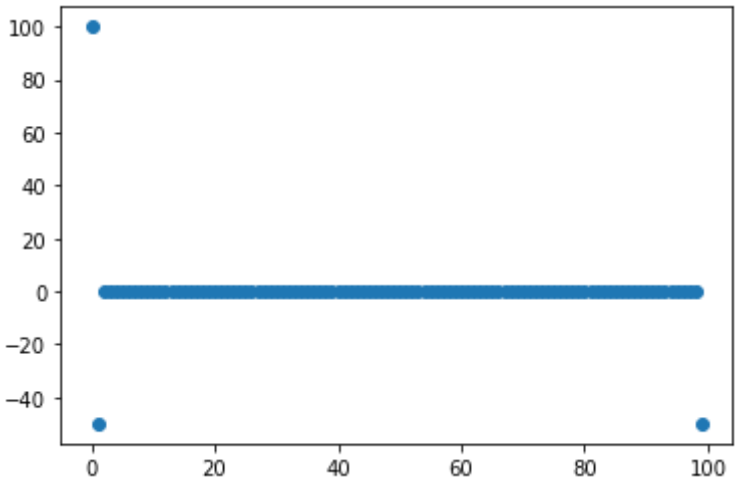
The width of spectral leakage is indeed greatly reduced. Note that the power is conserved, since the peak of the narrower blue curve is greater than the wider orange curve.

e)

Since cos is just a sum of two exponentials with definite modes ± 1, and the DFT of a

constant is simply a delta function centred at 0, we have that the DFT of our window is $W(k) = N/2\delta(k) - N/4\delta(k-1) - N/4\delta(k+1)$. Numerically:

In [299…
```
plt.scatter(ks,np.real(np.fft.fft(window)))
plt.show()
```



Indeed, the first term is $N/2 = 50$, the second is $-N/4 = -25$ and the last is $-N/4 = -25$. All others are zero.

Because of this, we only need to use 3 points in our original array we would like to Fourier transform. From the convolution theorem

$$F(f * W) = \tilde{f} \times \tilde{W} = N\tilde{f}\,(k)(\delta(k)/2 - \delta(k-1)/4 - \delta(k+1)/4) \qquad (6)$$

with W as our window. Since $\delta(k-\alpha)$ is a shift of α in k-space, we can use our previously built shifting function to write the windowed FFT:

In [300…
```
analytic_window_ft = num_sin/2 - shift_conv(num_sin,+1)/4 - shift_conv(num_sin,
# compare it to the numerical result
np.sum(analytic_window_ft - num_sin_adj)
```

Out[300]:
```
-1.5543122344752192e-15
```

Therefore, they are equivalent to within machine precision.

Question 5

Let's first get all the data from LIGO.

In [116…
```
# set directory with data files
ligo_dir = "LOSC_Event_tutorial-master/"
```

a)

Below are the functions we'll need to use to build a noise model. The main procedure goes

as follows:

- load a data file for a single event and a given detector
- dress a window with a tapered top so as not to ignore any events in the middle
- FT the windowed strain and smooth out its power spectrum with a Gaussian kernel
- take the noise model to be this smoothed out windowed strain
- since we assume white noise, the N^{-1} matrix has diagonals which are the inverse of the smoothed out noise.

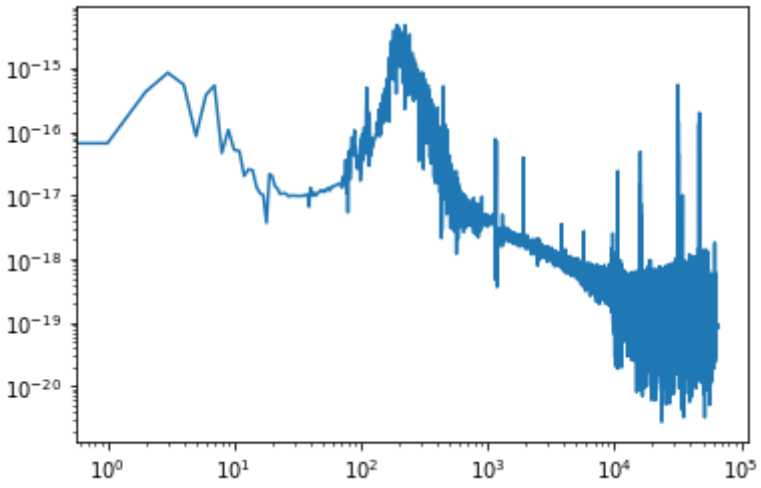
We then repeat this procedure for the other detector.

In [118…
```
import h5py
import glob

def smooth_vector(vec,sig):
    n=len(vec)
    x=np.arange(n)
#     x[n//2:]=x[n//2:]-n
    kernel=np.exp(-0.5*x**2/sig**2) #make a Gaussian kernel
    kernel=kernel/kernel.sum()
    vecft=np.fft.rfft(vec)
    kernelft=np.fft.rfft(kernel)
    vec_smooth=np.fft.irfft(vecft*kernelft) #convolve the data with the kernel
    return vec_smooth

def read_template(filename):
    dataFile=h5py.File(filename,'r')
    template=dataFile['template']
    tp=template[0]
    tx=template[1]
    return tp,tx
def read_file(filename):
    dataFile=h5py.File(filename,'r')
    dqInfo = dataFile['quality']['simple']
    qmask=dqInfo['DQmask'][...]

    meta=dataFile['meta']
    #gpsStart=meta['GPSstart'].value
    gpsStart=meta['GPSstart'][()]
    #print meta.keys()
    #utc=meta['UTCstart'].value
    utc=meta['UTCstart'][()]
    #duration=meta['Duration'].value
    duration=meta['Duration'][()]
    #strain=dataFile['strain']['Strain'].value
    strain=dataFile['strain']['Strain'][()]
    dt=(1.0*duration)/len(strain)

    dataFile.close()
    return strain,dt,utc

# test
strain,dt,utc = read_file(ligo_dir + 'H-H1_LOSC_4_V2-1126259446-32.hdf5')
```

Look at the strain from a given event:

In [4]:
```
strain_ft = np.fft.rfft(strain)
plt.loglog(np.abs(strain_ft))
```

Out[4]:
```
[<matplotlib.lines.Line2D at 0x7f9b5ae721c0>]
```



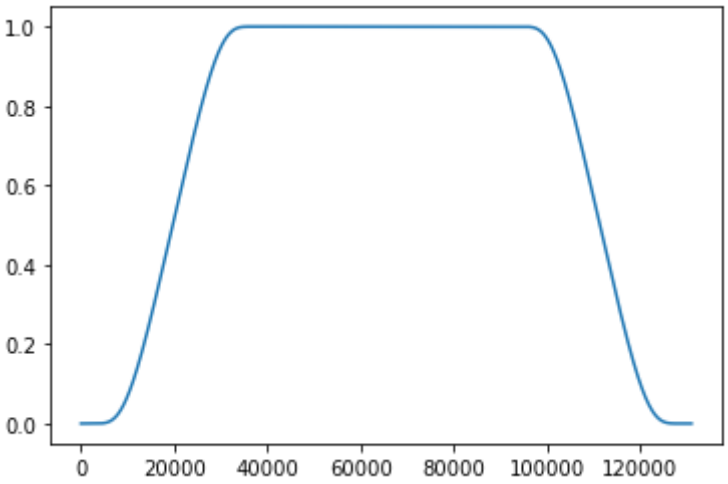
Looks like what we found in class.

First, the Hanford model.

Let's define a tapered window.

In [237…
```
# window
N = len(strain)

# tapered window
def tapered_window(N,eps):
    x = np.arange(N)
    window = (1+np.exp(eps*N/x-eps*N/(eps*N-x)))**(-1)
    window[0] = 0
    window[int(eps*N):int(N/2)] = 1
    window[int(N/2):] = np.flip(window[:int(N/2)])
    return window

window = tapered_window(N,eps=0.3)
plt.plot(window)
```

Out[237]:
```
[<matplotlib.lines.Line2D at 0x7f9b6d6d2ac0>]
```

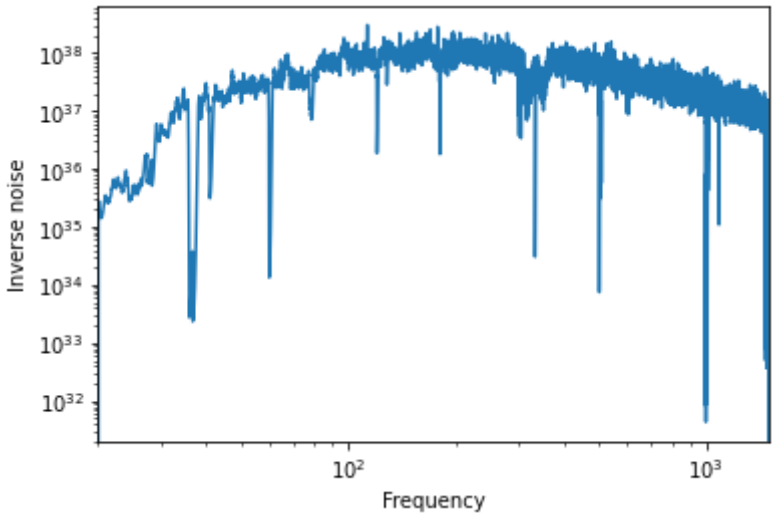


Look at the .json file to map each event to its template.

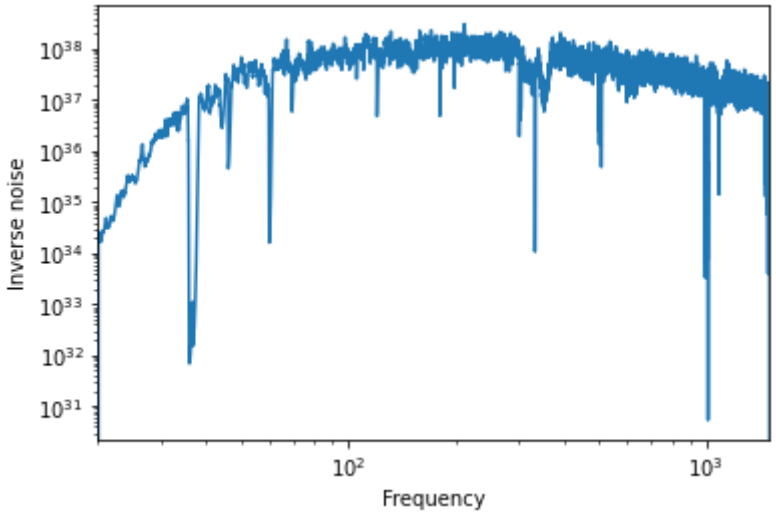
In [225…
```
fnames_H = ['H-H1_LOSC_4_V2-1126259446-32.hdf5',
            'H-H1_LOSC_4_V1-1167559920-32.hdf5',
            'H-H1_LOSC_4_V2-1128678884-32.hdf5',
            'H-H1_LOSC_4_V2-1135136334-32.hdf5']

fnames_L = ['L-L1_LOSC_4_V2-1126259446-32.hdf5',
            'L-L1_LOSC_4_V1-1167559920-32.hdf5',
            'L-L1_LOSC_4_V2-1128678884-32.hdf5',
            'L-L1_LOSC_4_V2-1135136334-32.hdf5']

templates = ['GW150914_4_template.hdf5',
             'GW170104_4_template.hdf5',
             'LVT151012_4_template.hdf5',
            'GW151226_4_template.hdf5']
```

In [239…
```
sig = 10  # sig = 10 is art more than science
eps = 0.3 # ditto

noise_profiles_H = np.zeros((len(fnames_H),N//2+1)) # where we keep our Ninv's

for i in range(len(fnames_H)):
    fname = fnames_H[i]
    fname = ligo_dir + fname
    print('Event:',fname)
    strain,dt,utc=read_file(fname)

    # window
    window = tapered_window(N,eps)

    # noise model
    noise_ft=np.fft.fft(window*strain)
    noise_smooth=smooth_vector(np.abs(noise_ft)**2,sig) # smooth out noise by c
    noise_smooth=noise_smooth[:len(noise_ft)//2+1] # will give us same length

    # compute the frequencies
    # we do this in the loop but we will use them later on as well
    tobs = dt * len(strain)
    dnu = 1/tobs
    nu = np.arange(len(noise_smooth))*dnu
    nu[0] = 0.5*nu[1]

    # the noise matrix is
    Ninv=1/noise_smooth
    Ninv[nu>1500]=0 # by inspection
    Ninv[nu<20]=0

    noise_profiles_H[i] = Ninv

    plt.clf()
    plt.loglog(nu,Ninv)
    plt.ylabel("Inverse noise")
    plt.xlabel("Frequency")
    plt.xlim(20,1500)
    plt.show()
    plt.close()
```

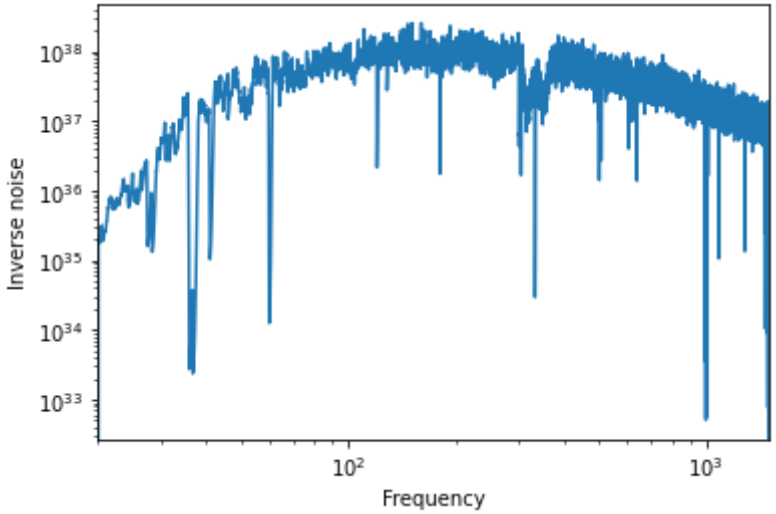
```
Event: LOSC_Event_tutorial-master/H-H1_LOSC_4_V2-1126259446-32.hdf5
```



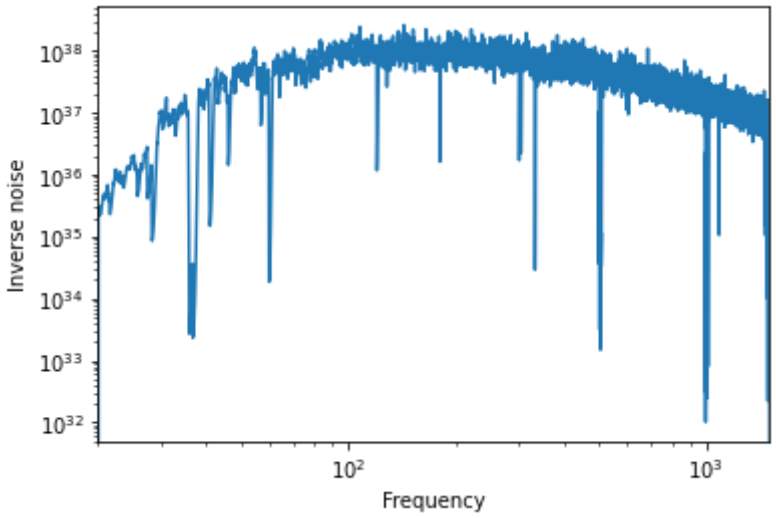

```
Event: LOSC_Event_tutorial-master/H-H1_LOSC_4_V1-1167559920-32.hdf5
```




```
Event: LOSC_Event_tutorial-master/H-H1_LOSC_4_V2-1128678884-32.hdf5
```




```
Event: LOSC_Event_tutorial-master/H-H1_LOSC_4_V2-1135136334-32.hdf5
```

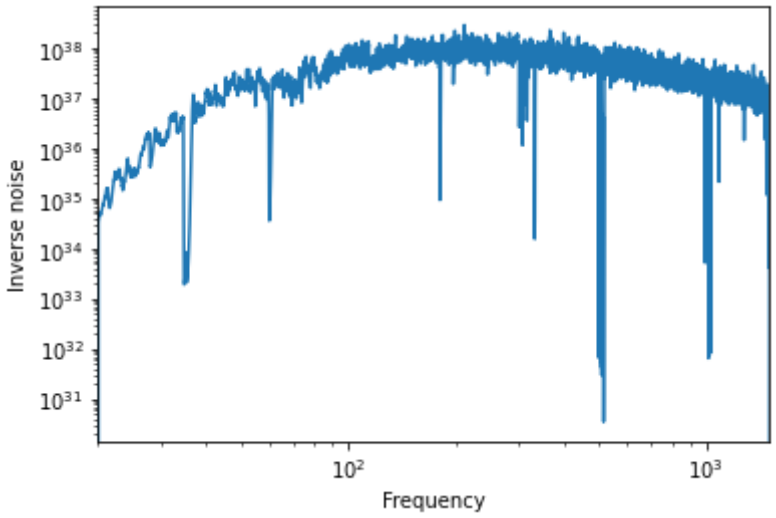


Now, the Livingston detector:

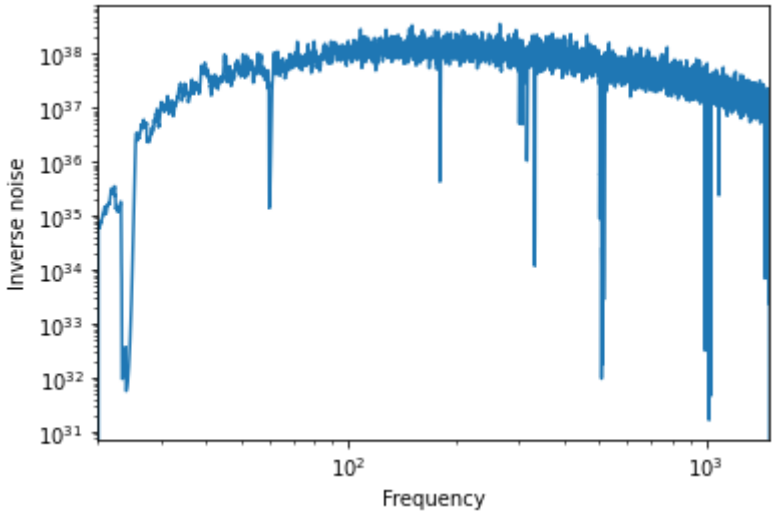
In [240…

```
sig = 10  # sig = 10 is art more than science
eps = 0.3

noise_profiles_L = np.zeros((len(fnames_H),N//2+1))

for i in range(len(fnames_L)):
    fname = fnames_L[i]
    fname = ligo_dir + fname
    print('Event:',fname)
    strain,dt,utc=read_file(fname)

    # window
    window = tapered_window(N,eps)

    # noise model
    noise_ft=np.fft.fft(window*strain) # window it
    noise_smooth=smooth_vector(np.abs(noise_ft)**2,sig) # smooth it out
    noise_smooth=noise_smooth[:len(noise_ft)//2+1] # will give us same length

    tobs = dt * len(strain)
    dnu = 1/tobs
    nu = np.arange(len(noise_smooth))*dnu
    nu[0] = 0.5*nu[1]

    # the noise matrix is
    Ninv=1/noise_smooth
    Ninv[nu>1500]=0 # by inspection
    Ninv[nu<20]=0

    noise_profiles_L[i] = Ninv

    plt.clf()
    plt.loglog(nu,Ninv)
    plt.ylabel("Inverse noise")
    plt.xlabel("Frequency")
    plt.xlim(20,1500)
    plt.show()
    plt.close()
```

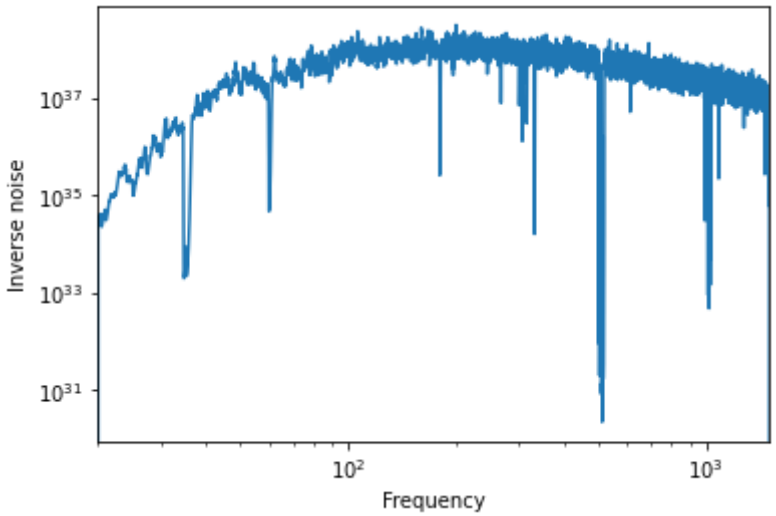
```
Event: LOSC_Event_tutorial-master/L-L1_LOSC_4_V2-1126259446-32.hdf5
```



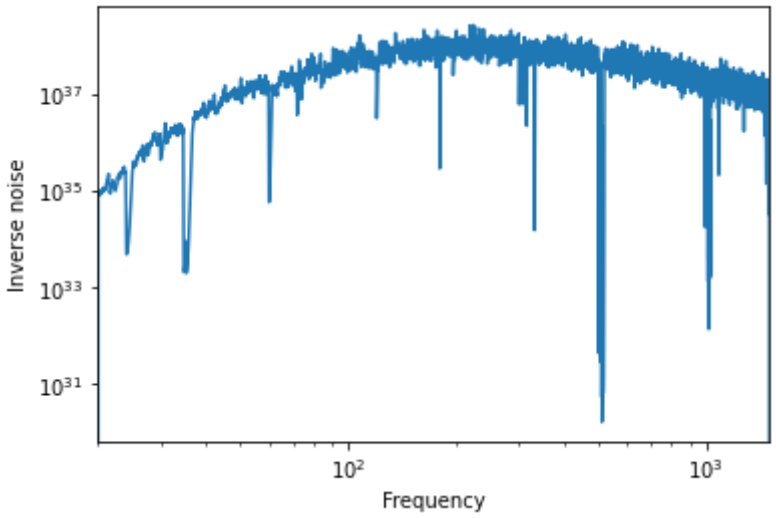

```
Event: LOSC_Event_tutorial-master/L-L1_LOSC_4_V1-1167559920-32.hdf5
```




```
Event: LOSC_Event_tutorial-master/L-L1_LOSC_4_V2-1128678884-32.hdf5
```




```
Event: LOSC_Event_tutorial-master/L-L1_LOSC_4_V2-1135136334-32.hdf5
```

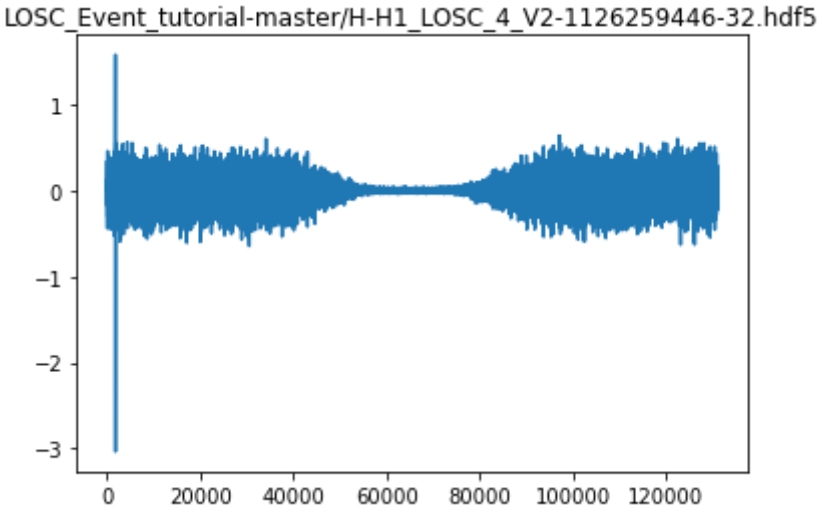


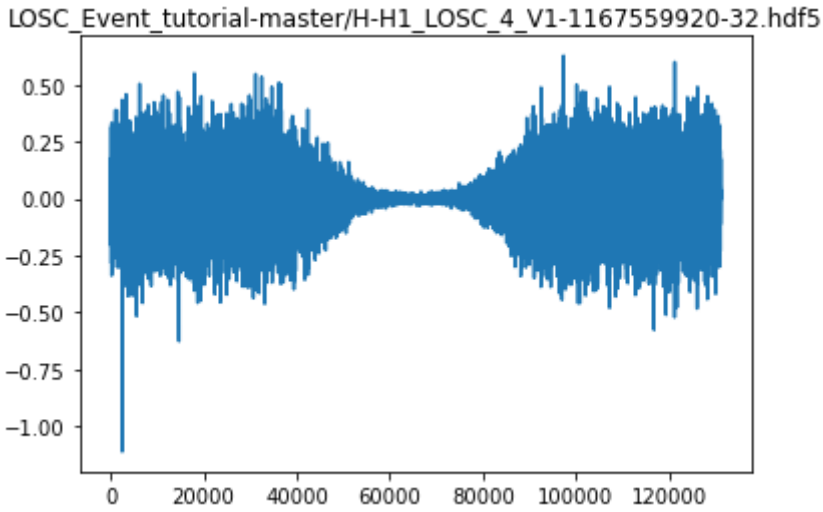
b)

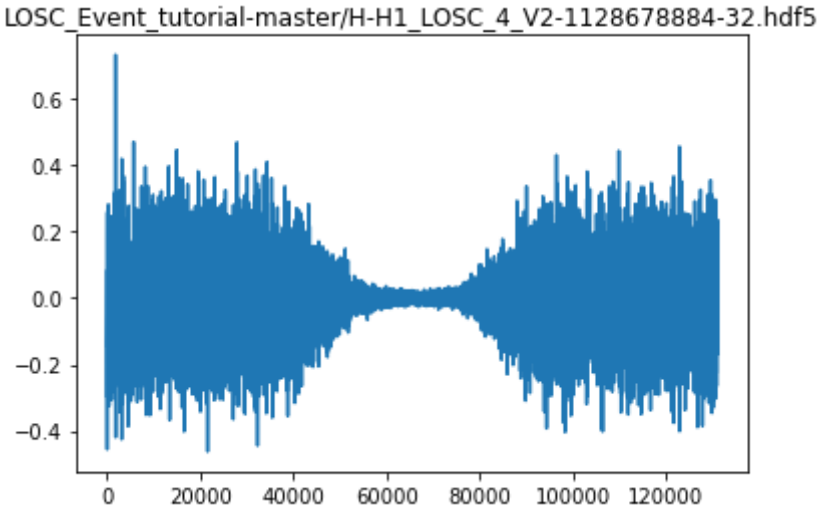
We would now like to search each event using a matched filter. This amounts to calculating the correlation between the template and strain, taking our noise model into account. We will do this once for each of the three possible templates.

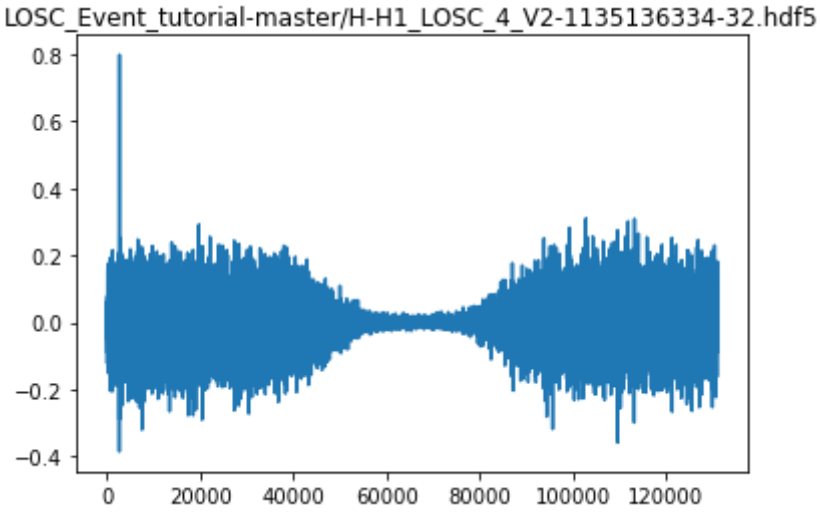
In [241…

```
# loop over each event
rhs_H = np.zeros((len(fnames_H),N))

for i in range(len(fnames_H)):
    # set template
    temp = ligo_dir + templates[i]
    tp,tx = read_template(temp) # we'll use the plus polarization

    # move to fourier space
    template_ft=np.fft.rfft(tp*window)

    # multiply by the noise model
    template_filt=template_ft*noise_profiles_H[i]

    # move strain to fourier space as well
    fname = ligo_dir + fnames_H[i]
    strain,dt,utc=read_file(fname)
    window = tapered_window(N,eps)

    data_ft=np.fft.rfft(strain*window)

    # math filter
    rhs=np.fft.irfft(data_ft*np.conj(template_filt)) # lhs is identity by const
    rhs_H[i] = rhs

    plt.title(fname)
    plt.plot(rhs)
    plt.show()
```


LOSC_Event_tutorial-master/H-H1_LOSC_4_V2-1126259446-32.hdf5
1
0
−1
−2
−3
0
20000
40000
60000
80000
100000
120000

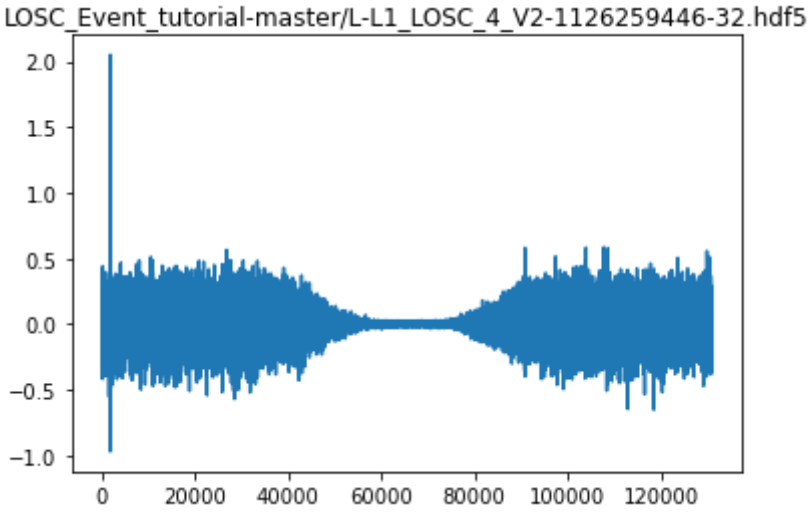

LOSC_Event_tutorial-master/H-H1_LOSC_4_V1-1167559920-32.hdf5
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
20000
40000
60000
80000
100000
120000

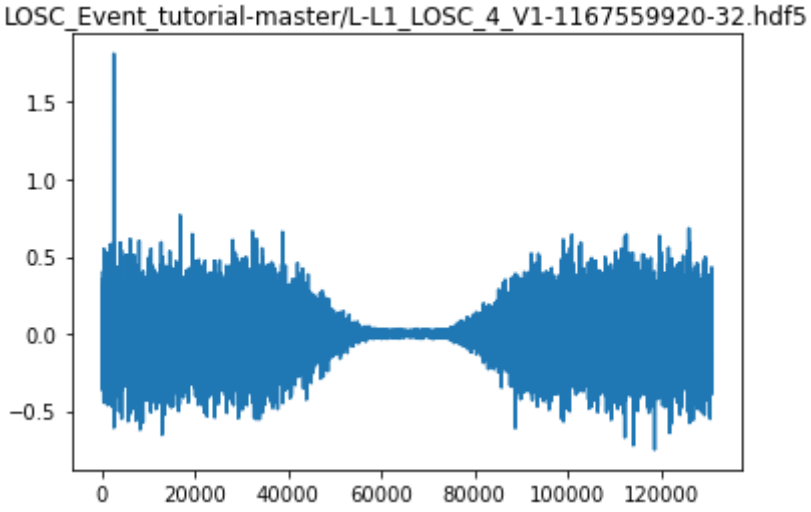

LOSC_Event_tutorial-master/H-H1_LOSC_4_V2-1128678884-32.hdf5
0.6
0.4
0.2
0.0
−0.2
−0.4
0
20000
40000
60000
80000
100000
120000

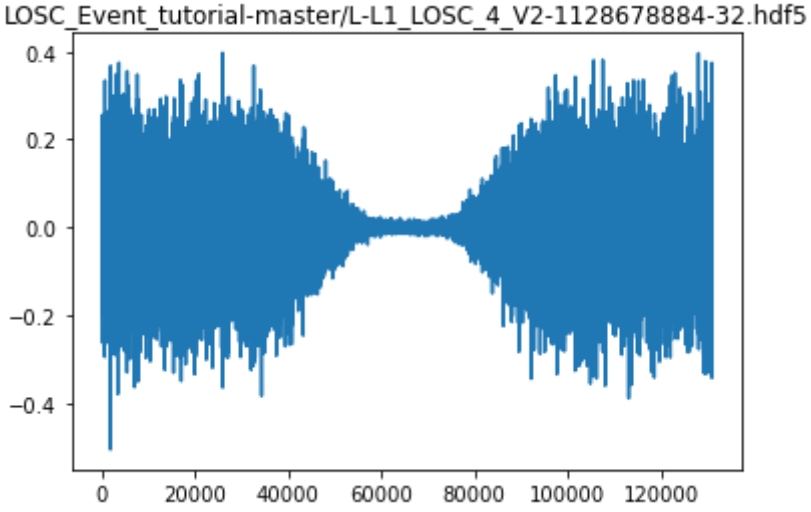


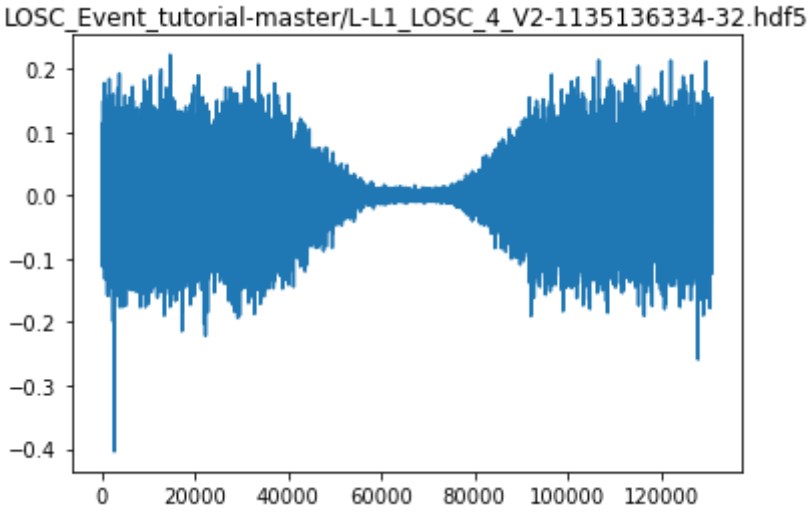
In [242…

```
# loop over each event
rhs_L = np.zeros((len(fnames_H),N))

for i in range(len(fnames_L)):
    # set template
    temp = ligo_dir + templates[i]
    tp,tx = read_template(temp) # we'll use the plus polarization

    # move to fourier space
    template_ft=np.fft.rfft(tp*window)

    # multiply by the noise model
    template_filt=template_ft*noise_profiles_L[i] ### this is the PS : abs(FT(t

    # move strain to fourier space as well
    fname = ligo_dir + fnames_L[i]
    strain,dt,utc=read_file(fname)
    window = tapered_window(N,eps)

    data_ft=np.fft.rfft(strain*window)

    # math filter
    rhs=np.fft.irfft(data_ft*np.conj(template_filt)) # lhs is identity by const
    rhs_L[i] = rhs

    plt.title(fname)
    plt.plot(rhs)
    plt.show()
```


LOSC_Event_tutorial-master/L-L1_LOSC_4_V2-1126259446-32.hdf5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
0
20000
40000
60000
80000
100000
120000


LOSC_Event_tutorial-master/L-L1_LOSC_4_V1-1167559920-32.hdf5
1.5
1.0
0.5
0.0
−0.5
0
20000
40000
60000
80000
100000
120000


LOSC_Event_tutorial-master/L-L1_LOSC_4_V2-1128678884-32.hdf5
0.4
0.2
0.0
−0.2
−0.4
0
20000
40000
60000
80000
100000
120000



The gravitational wave events are clearly visible in each plot (the spike near 0 frequency).

In [91]:
```
ps = np.abs(np.fft.rfft(rhs))**2
```

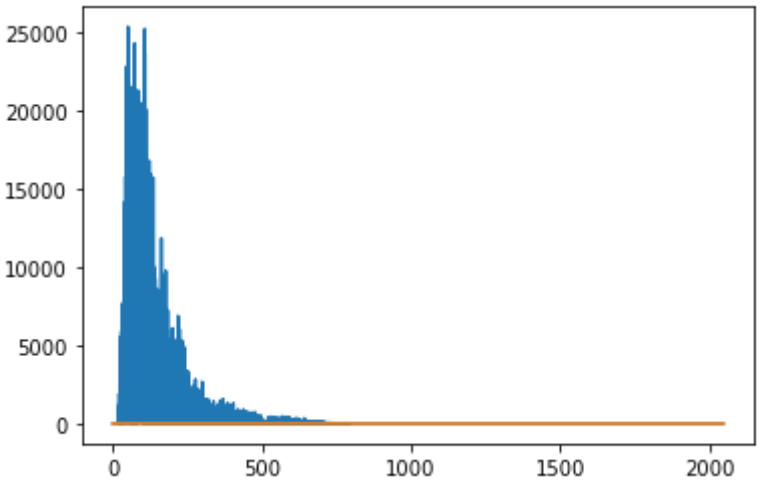
In [92]:
```
tobs = dt * len(strain)
dnu = 1/tobs
nu = np.arange(len(noise_smooth))*dnu
nu[0] = 0.5*nu[1]
```

In [94]:
```
np.sum(ps*nu)/np.sum(ps)
```

Out[94]:
```
143.17891823047242
```

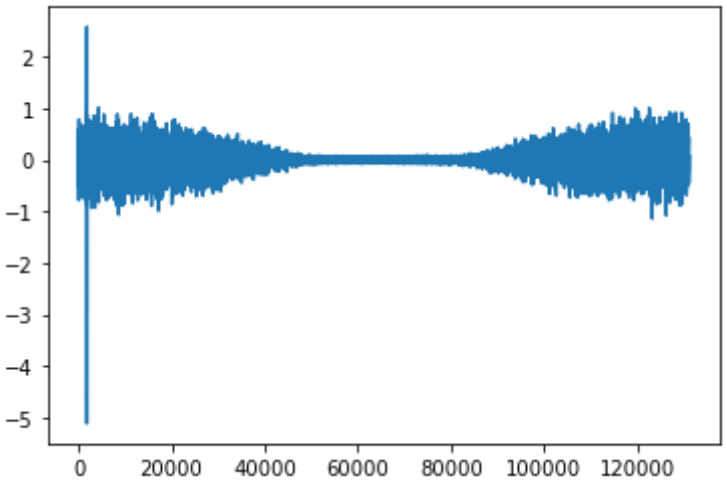
In [102…
```
plt.plot(nu,ps)
plt.plot(nu,ps_cum)
```

Out[102]:
```
[<matplotlib.lines.Line2D at 0x7f9b5c317df0>]
```



In [233…
```
plt.plot(rhs_H[0])
```

Out[233]:
```
[<matplotlib.lines.Line2D at 0x7f9b094f8430>]
```



In [234…
```
np.mean(np.abs(rhs_H[0]))
```

Out[234]:
```
0.12850813500544203
```

In [235…
```
np.max(np.abs(rhs_H[0])) / np.mean(np.abs(rhs_H[0]))
```

Out[235]:
```
39.83176949700623
```

c)

Let's estimate the SNR both with noise computed numerically (by looking at the MF scatter) and analytically (by computing $A^T N^{-1} A$).

In [247…
```
# i had to be a bit clever for this bit
# because my machine couldn't matrix multiply arrays this large (not sure why)
# so i filtered them in multiples of 10, noting that the more i filtered the mo
# A.T@Ninv@A was reduced by a factor of sqrt(10)
# i therefore extrapolated that the true value should be 10 times what i found
# it's a bit sloppy, but as you can see it is (miraculously) somewhat in agreem
# sorry for the ad-hoc-ness

# loop over a given event
# do both H and L detectors
for i in range(len(fnames_H)):
    # read in templates again
    temp = ligo_dir + templates[i]
    tp,tx = read_template(temp)

    # this is the noise
    temp_red = tp[:len(tp)//2+1][::10]
    Ninv_red_H = noise_profiles_H[i][::10]
    Ninv_red_L = noise_profiles_L[i][::10]
    noise_ana_H = 1/np.sqrt(temp_red.T@np.diag(Ninv_red_H)@temp_red) * np.sqrt(
    noise_ana_L = 1/np.sqrt(temp_red.T@np.diag(Ninv_red_L)@temp_red) * np.sqrt(

    # estimate of the noise from scatter
    # do it the usual way: either std(__) or mean(abs(__))
    noise_num_H = np.mean(np.abs(rhs_H[i]))
    noise_num_L = np.mean(np.abs(rhs_L[i]))
```

```
    # this is the signal
    # simply the max of the scatter array
    signal_H = np.max(np.abs(rhs_H[i]))
    signal_L = np.max(np.abs(rhs_L[i]))

    # take the ratio to get the SNR
    snr_ana_H = signal_H / noise_ana_H
    snr_num_H = signal_H / noise_num_H
    snr_ana_L = signal_L / noise_ana_L
    snr_num_L = signal_L / noise_num_L

    # the combined SNR is simply added in quadrature
    snr_num_combined = np.sqrt(snr_num_H**2 + snr_num_L**2)
    snr_ana_combined = np.sqrt(snr_ana_H**2 + snr_ana_L**2)

    # print for clarity
    print("Event: {}".format(fnames_H[i]))
    print("The estimated SNR at Hanford detectors is {}".format(snr_num_H))
    print("The analytical SNR at Hanford detectors is {}".format(snr_ana_H))
    print("The estimated SNR at Livingston detectors is {}".format(snr_num_L))
    print("The analytical SNR at Livingston detectors is {}".format(snr_ana_L))
    print("The estimated combined SNR is {}".format(snr_num_combined))
    print("The analytical combined SNR is {}".format(snr_ana_combined))
    print("\n")
```

```
Event: H-H1_LOSC_4_V2-1126259446-32.hdf5
The estimated SNR at Hanford detectors is 30.46203862713007
The analytical SNR at Hanford detectors is 35.02241574302264
The estimated SNR at Livingston detectors is 23.393546849873044
The analytical SNR at Livingston detectors is 24.342803971037267
The estimated combined SNR is 38.40825212812957
The analytical combined SNR is 42.65139751109532


Event: H-H1_LOSC_4_V1-1167559920-32.hdf5
The estimated SNR at Hanford detectors is 13.203831155764965
The analytical SNR at Hanford detectors is 10.559416353632386
The estimated SNR at Livingston detectors is 17.177009198080484
The analytical SNR at Livingston detectors is 19.174580299318002
The estimated combined SNR is 21.66542873291205
The analytical combined SNR is 21.889856175506342


Event: H-H1_LOSC_4_V2-1128678884-32.hdf5
The estimated SNR at Hanford detectors is 10.85671699808926
The analytical SNR at Hanford detectors is 4.502157929334679
The estimated SNR at Livingston detectors is 8.264550975055647
The analytical SNR at Livingston detectors is 3.3297740282905104
The estimated combined SNR is 13.644453334446695
The analytical combined SNR is 5.599716162462981


Event: H-H1_LOSC_4_V2-1135136334-32.hdf5
The estimated SNR at Hanford detectors is 17.181890002547505
The analytical SNR at Hanford detectors is 2.226956821384972
The estimated SNR at Livingston detectors is 12.43488946064992
The analytical SNR at Livingston detectors is 1.126864639749456
The estimated combined SNR is 21.209521917248026
The analytical combined SNR is 2.495828640077425
```

d)

There is a lot of discrepancy between both SNRs. This has to due with sources of error in both the numerical estimates and analytical template MF.

First, the analytical results. Since I could not obtain noise from my template matched filter (regrettably, my machine's kernel died when I tried to matrix multiply these matrices), I had to resort to tricks to arbitrarily reduce my data set. This leads to a wide disparity in the obtained results, though they are (thankfully) on the same order of magnitude (for the most part).

Another potential source of error is the choice of a window function. By choosing a tapered window function, we effectively let the data at the edges drop off exponentially quickly (which would not have happened had we chosen a cos window function, for instance). That said, a lot of our data is exponentially suppressed and close to zero as evidenced by the correlation function/MF scatter plots. Therefore, the numerical estimate of noise, be it through `np.std` or `np.mean(np.abs())` will always under-estimate the noise, thus

inflating the SNR. This is why the numerical SNR is always much greater than the analytical one, though the analytical one is also erroneous for the reasons listed above.

e)

The noise power spectrum is simply the FT of the correlation function, or the whitened template spectrum. It has the form $F(temp)^2/N$:

In [249…

```
# loop over each event
ps_H = np.zeros((len(fnames_H),N//2+1))
ps_L = np.zeros((len(fnames_H),N//2+1))
nu_mid_H = np.zeros(len(fnames_H))
nu_mid_L = np.zeros(len(fnames_H))

for i in range(len(fnames_H)):
    # load template
    temp = ligo_dir + templates[i]
    tp,tx = read_template(temp)

    # FT the template
    template_ft = np.fft.rfft(tp*window)

    # get the power spectrum
    # the power spectrum is FT(temp)^2 / N
    # alternatively, just the FT of the correlation function
    ps_H[i] = np.abs(template_ft)**2 * noise_profiles_H[i]
    ps_L[i] = np.abs(template_ft)**2 * noise_profiles_L[i]

    # find the frequency that divides the array into two equally-weighted parts
    # this is equivlanet to finding the Centre of mass
    nu_mid_H[i] = np.sum(nu*ps_H[i])/np.sum(ps_H[i])
    nu_mid_L[i] = np.sum(nu*ps_L[i])/np.sum(ps_L[i])

    # print for clarity
    print("Event: {}".format(fnames_H[i]))
    print("Frequency where half the weight comes from below (Hanford): {}".form
    print("Frequency where half the weight comes from below (Livingston): {}".f
    print("\n")
```

```
Event: H-H1_LOSC_4_V2-1126259446-32.hdf5
Frequency where half the weight comes from below (Hanford): 115.60834913217806
Frequency where half the weight comes from below (Livingston): 125.68980307057
346


Event: H-H1_LOSC_4_V1-1167559920-32.hdf5
Frequency where half the weight comes from below (Hanford): 115.37628335041603
Frequency where half the weight comes from below (Livingston): 98.262492236222
9


Event: H-H1_LOSC_4_V2-1128678884-32.hdf5
Frequency where half the weight comes from below (Hanford): 100.90584383150572
Frequency where half the weight comes from below (Livingston): 115.91205970646
38


Event: H-H1_LOSC_4_V2-1135136334-32.hdf5
Frequency where half the weight comes from below (Hanford): 104.05849427680106
Frequency where half the weight comes from below (Livingston): 142.61269206043
437
```

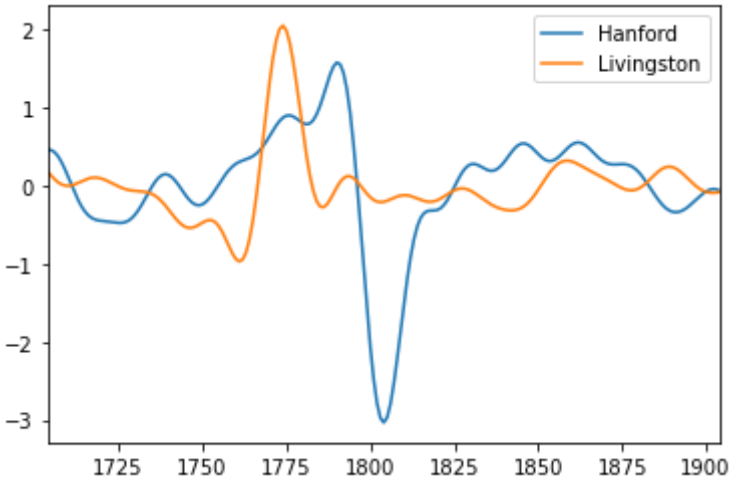
f)

We are now in a position to compute the time-shift between an event at both detectors.

To see how well we can localize the time of arrival, let's zoom in on a single matched filter correlation function:

In [267…

```
idx = np.argmax(np.abs(rhs_H[0]))

plt.plot(rhs_H[0],label="Hanford")
plt.plot(rhs_L[0],label="Livingston")
plt.xlim(idx-100,idx+100)
plt.legend()
plt.show()
```



As we can see, the width of the peak is roughly $20 \times dt$ seconds acroos, so we can localize

the time of arrival to roughly

In [272…
```
20 * dt
```

Out[272]:
```
0.0048828125
```

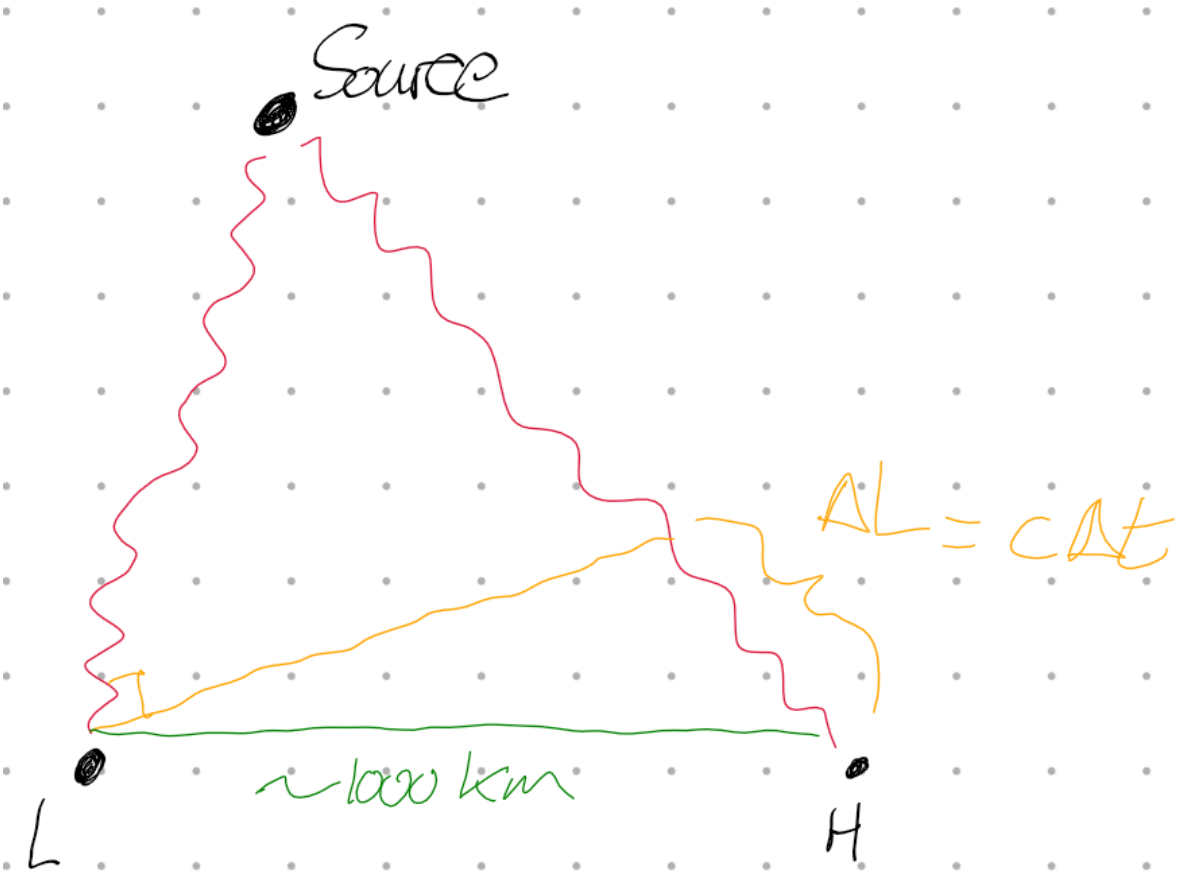
seconds. More importantly, the width of one peak is narrower than the distance between peaks, allowing us to gauge the time shift.

We must now compare the times in both the Hanford and Livingston events to obtain the shift Δt:

In [273…
```
delta = np.argmax(np.abs(rhs_H[0])) - np.argmax(np.abs(rhs_L[0]))
delta_t = delta * dt
print(delta_t)
```

```
0.00732421875
```

To pinpoint a gravitational wave, we need its angle above the horizon. We use the diagram posted in the slack :



We see that the angle above the horizon is given by $\theta = \pi/2 + \arctan c\Delta t/L$, with L the distance between detectors. The uncertainty on this angle is therefore $\delta\theta = \frac{c\delta t/L}{1+(c\Delta t/L)^2}$ with δt the error in the events, i.e. the width of the peaks. Plugging in our numbers gives:

In [306…
```
del_t = 20 * dt
L = 1e6
```

```
alpha = 3e8*delta_t/L
```

In [307…
```
3e8 * del_t / (1 + alpha**2) / L
```

Out[307]:
```
0.2513468986476359
```

which is our estimate for the positional uncertainty of a gravitational wave event.

In [ ]:
```

```