1 Question 1

To shift continuous data with a convolution, a shifted delta function kernel is used (where the delta is the kernel of the identity convolution, and a shift in the kernel produces a shift in the convolution result.) For the discrete case, consider these discrete convolutions:

$$\sum_{x'=0}^{N} f[x']g_0[x-x'] = f[x]\, g_0[y] = f[x]$$

$$\sum_{x'=0}^{N} f[x']g_1[x-x'] = f[x+1]\, g_1[y+1] = f[x+1]$$

$$\vdots$$

$$\sum_{x'=0}^{N} f[x']g_n[x-x'] = f[x+n]\, g_n[y+n] = f[x+n]$$

where y is the only index of the array g_0 where its value is 1 and not 0. I.e. the convolution of f with g_0 at index x 'picks out' the value of f at index x. Therefore this g_0 vector is the discrete convolution identity kernel. Similarly, g_1 and g_n are the analogues of the shifted delta function kernels, where the non-zero indices are $y+1$ and $y+n$ respectively.

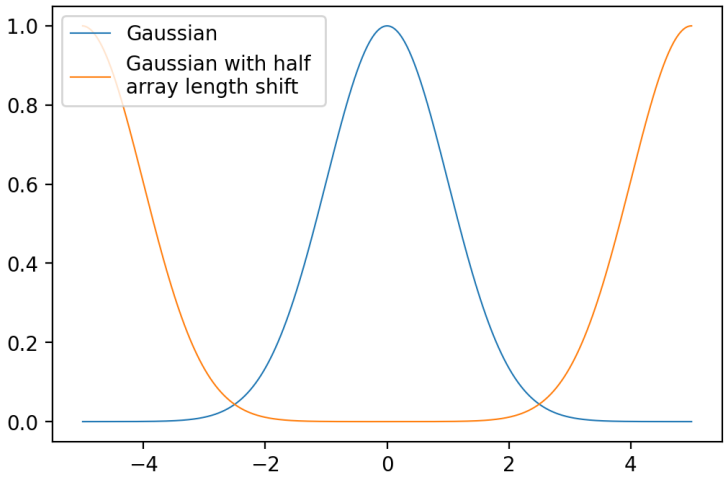
For odd N, the index where vector g_0 is non-zero is just the middle index. For even N, this index was found to be the greater of the two middle indices. Therefore it is easy to produce any g_n by taking g_0 and shifting its non-zero index by n.

Here is the code for taking the convolution via FFT and then for shifting an array using this FFT convolution with the g_n vector described above:

```
def convolve(f,g):
    f_ft=np.fft.fft(f)
    g_ft=np.fft.fft(g)
    conv=np.fft.ifft(f_ft*g_ft)
    return np.fft.fftshift(conv)  # FFT shift for negative frequencies

def conv_shift(x,n):
    size=len(x)
    delta_shift=np.zeros(size)
    delta_shift[int(size/2)+n]=1  # int(size/2) index works for even or odd size
    return convolve(x,delta_shift)

N=2000
x=np.linspace(-5,5,N)
y=gaussian(x,0,1)
y_shift=conv_shift(y,N//2-1)
plt.plot(x,y)
plt.plot(x,y_shift)
```

Since 0 is the first index, for even N, `int(size/2)` gives the greater of the two middle indices. For odd N, `int(size/2)` rounds down, giving the middle index.

Here is the result of a shifted Gaussian (next page):



2 Question 2

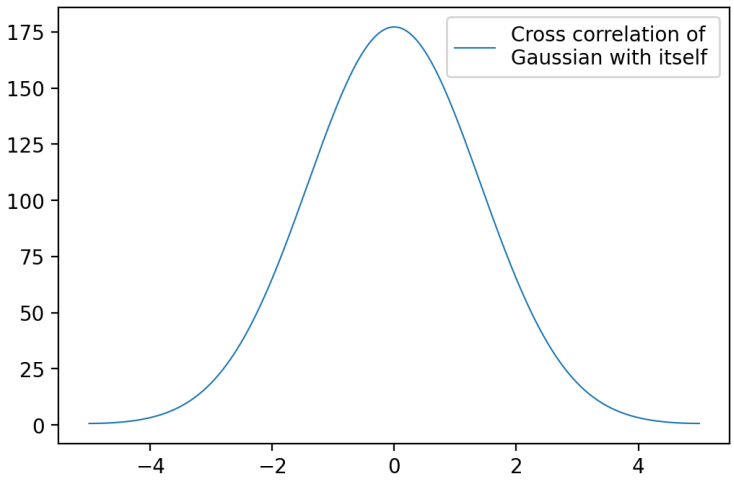
a)

To calculate the correlation of two arrays, the following function was written:

```
def correlation(f,g):
    f_ft=np.fft.fft(f)
    g_ft=np.fft.fft(g)
    corr=np.fft.ifft(f_ft*np.conj(g_ft))
    return np.fft.fftshift(corr)
```

The correlation of a Gaussian with itself was calculated and plotted:

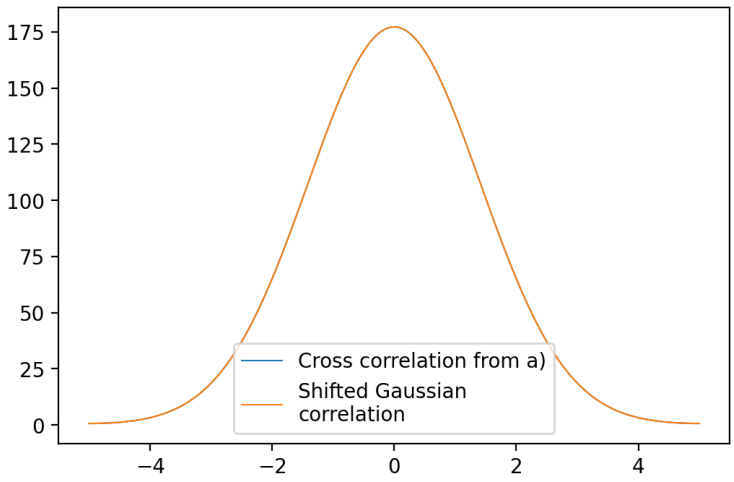
```
x=np.linspace(-5,5,1001)
y=gaussian(x,0,1)
corr=correlation(y,y)
plt.plot(x,corr)
plt.legend(['Cross correlation of Gaussian with itself'])\
```



b)

Now to calculate and plot the correlation of a shifted Gaussian with itself and compare to the result from a):

```
y_shift=conv_shift(y,100)
shift_corr=correlation(y_shift,y_shift)
plt.plot(x,corr)
plt.plot(x,shift_corr)
```



There is no difference if both arrays being correlated are shited by the same amount. If we look at the convolution formula:

$$(f * g)(x) = \int_{-\infty}^{+\infty} f(y)g(x+y)dx'$$

The convolution of the two functions $f'(x) = f(x+\delta)$ and $g'(x) = g(x+\delta)$ is:

$$(f' * g')(x) = \int_{-\infty}^{+\infty} f(y+\delta)g(x+y+\delta)dy$$

We can change the integration variable to $y' = y + \delta$, and since y ranges over all values, the integration bounds are unchaged, giving:

$$(f' * g')(x) = \int_{-\infty}^{+\infty} f(y')g(x+y')dy'$$

$$\Rightarrow f' * g' = f * g$$

Therefore it is not a surprise that cross correlation between the shifted Gaussian and itself is the same as in part a).

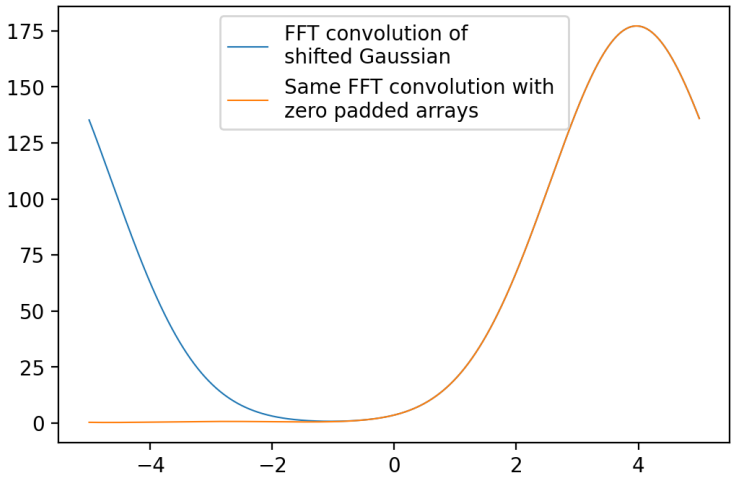
3 Question 3

To avoid the error at the edges of convolution due to the circulant nature of the DFT, the input arrays are first zero-padded on both ends. Since the arrays are now 0 at the end, the FFT produces less error at the endpoints since they are no longer forced to have the same value. After the padded arrays are convolved, the ends of the result array are removed in accordance with the number of zeros padding the input arrays. Here is the code:

```
def padded_conv(f,g,npad):
    pad=np.zeros(npad)
    fpad=np.append(pad,f)
    fpad=np.append(fpad,pad)
    gpad=np.append(pad,g)
    gpad=np.append(gpad,pad)
    conv_pad=convolve(fpad,gpad)
    return conv_pad[npad:-npad]
```

A shifted Gaussian was then convolved with itself using this function and compared to the non-padded result:

```
x=np.linspace(-5,5,1001)
y=gaussian(x,0,1)
yshift=conv_shift(y,200)
window=np.cos(np.pi*x/(x[-1]-x[0]))**2
conv=convolve(yshift,yshift)
conv_pad=padded_conv(yshift,yshift,200)
plt.plot(x,conv)
plt.plot(x,conv_pad)
```



The circulant nature of orignal FFT convolution is apparent on the left, while the zero-padded result does not have this error.

4 Question 4

a)

The partial sum for the geometric series is:

$$\sum_{i=0}^{N-1} \alpha^x = \frac{1-\alpha^N}{1-\alpha}$$

Using $\alpha = \exp(-2\pi ik/N)$:

$$\sum_{i=0}^{N-1} \exp(-2\pi ik/N)^x = \frac{1-\exp(-2\pi ik/N)^N}{1-\exp(-2\pi ik/N)} = \frac{1-\exp(-2\pi ik)}{1-\exp(-2\pi ik/N)}$$

b)

Taking the limit of the partial sum above:

$$\lim_{k\to 0}\frac{1-\exp(-2\pi ik)}{1-\exp(-2\pi ik/N)} =^{LHR} \lim_{k\to 0}\frac{\frac{d}{dk}[1-\exp(-2\pi ik)]}{\frac{d}{dk}[1-\exp(-2\pi ik/N)]} = \lim_{k\to 0}\frac{2\pi i\exp(-2\pi ik)}{(2\pi i/N)\exp(-2\pi ik/N)}$$

$$= \frac{2\pi i}{2\pi i/N} = N$$

If k is an integer, then:

$$1-\exp(-2\pi ik) = 0$$

Also, if k is not a multiple of N, k/N is not an integer and therefore:

$$1-\exp(-2\pi ik/N) \neq 0$$

$$\Rightarrow \frac{1-\exp(-2\pi ik)}{1-\exp(-2\pi ik/N)} = 0$$

c)

Writing the analytic sum for the DFT of a sine wave with frequency k_0:

$$F(k) = \sum_{x=0}^{N-1} f(x)\exp(-2\pi ik) = \sum_{x=0}^{N-1}\exp(2\pi ik_0x)\exp(-2\pi ikx) = \sum_{x=0}^{N-1}\exp[-2\pi i(k-k_0)x]$$

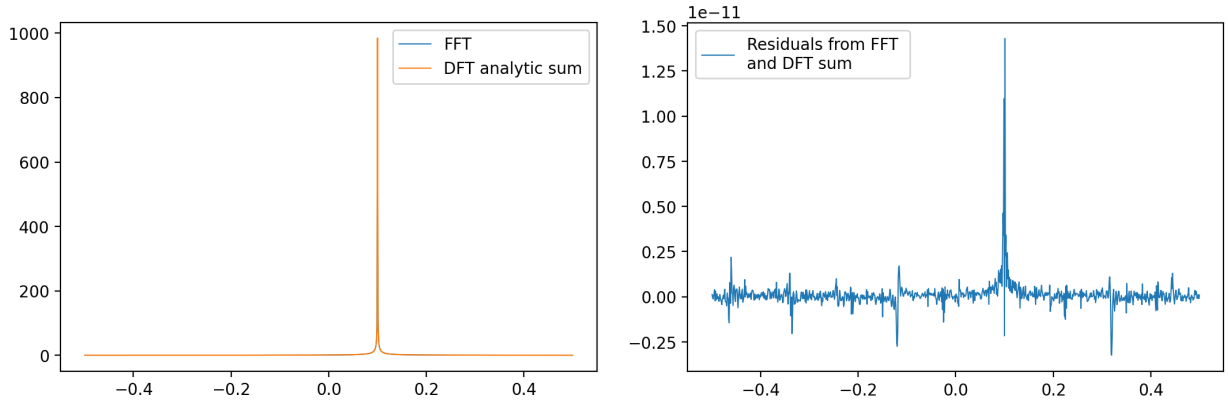
Using the result from a):

$$F(k) = \sum_{x=0}^{N-1}\exp[-2\pi iN(k-k_0)x/N] = \frac{1-\exp[-2\pi iN(k-k_0)]}{1-\exp[-2\pi iN(k-k_0)/N]} = \frac{1-\exp[2\pi iN(k_0-k)]}{1-\exp[2\pi i(k_0-k)]}$$

Using this code to compare the FFT vs. the DFT sum:

```
def dft_sum(k,k0,N):
    a=1-np.exp(-2J*np.pi*N*(k-k0))
    b=1-np.exp(-2J*np.pi*(k-k0))
    return a/b

k0=0.1
N=1001
x=np.linspace(0,N-1,N)
y=np.exp(2J*np.pi*k0*x)
ft=np.fft.fft(y)
k=np.fft.fftfreq(N,np.abs(x[1]-x[0]))
plt.plot(np.fft.fftshift(k),np.fft.fftshift(np.abs(ft)))  # fftshift to plot neg. freqs
plt.plot(np.fft.fftshift(k),np.fft.fftshift(np.abs(dft_sum(k,k0,N))))
```

The plots and the residuals are on the next page:



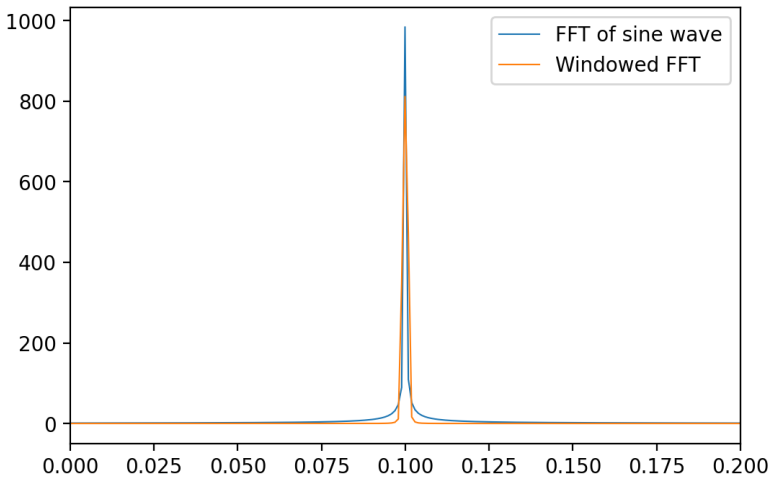
As seen in the plots above, the FFT and DFT sum are in agreement, with an error which is essentially just machine precision.

We can also see in the plot on the right the phenomenon of spectral leakage. We expect a delta function when Fourier transforming a pure frequency sine wave, however since we are taking a discrete Fourier transform, we expect a non-perfect delta spike with some finite width. In this case, the spike has a small width but is otherwise quite close do a delta function.

d)

Here is the code used for the windowed Fourier transform of a pure frequency sine wave and the resulting plot:

```
def window_fun(x,N):
    return 0.5-0.5*np.cos(2*np.pi*x/N)

ft_wndw=np.abs(np.fft.fft(y*window_fun(x,N)))
norm=np.sqrt(np.mean(window_fun(x,N)**2))
ft_wndw=ft_wndw/norm
ft_wndw_shift=np.fft.fftshift(ft_wndw)
plt.plot(kshift,ftshift)
plt.plot(kshift,ft_wndw_shift)
plt.xlim([0,0.2])
```



As seen in the plot above, the spectral leakage drops quite dramatically, with sharper corners at the bottom of the spike instead of the gradual decay that leaks into nearby frequencies.

e)

Printing the FFT of this window function (N=1001) in python gives:

$$[5.00500000e{+}02\ -2.50250000e{+}02\ 1.45623712e{-}14\ \ldots\ 4.09901694e{-}15\ 1.45623712e{-}14\ -2.50250000e{+}02]$$

$$= [\mathrm{N}/2\ -\mathrm{N}/4\ 0\ \ldots\ 0\ -\mathrm{N}/4]$$

Where if we take the absolute value (for negative frequencies) we get:

$$[\mathrm{N}/2\ \mathrm{N}/4\ 0\ \ldots\ 0\ \mathrm{N}/4]$$

Since the multiplication of the window in real space corresponds to a convolution in Fourier space, we will consider a discrete convolution:

$$(F * G)(k) = \int_{-\infty}^{\infty} F(k')G(k-k')dk' \ \rightarrow\ (F * G)[k] = \sum_{k'=-\infty}^{k'=+\infty} F[k']G[k-k']$$

If $G[k]$ is the window function $[N/2, -N/4, 0, ..., 0, -N/4]$:

$$G[k-k'] \neq 0 \ \Rightarrow\ k-k' = -1, 0, 1 \ \Rightarrow\ k' = k-1,\ k\ , k+1$$

Therefore the sum reduces to three terms:

$$(F * G)[k] = F[k-1]G[-1] + F[k]G[0] + F[k+1]G[1]$$

$$= -\frac{N}{4}F[k-1] + \frac{N}{2}F[k] - \frac{N}{4}F[k+1]$$

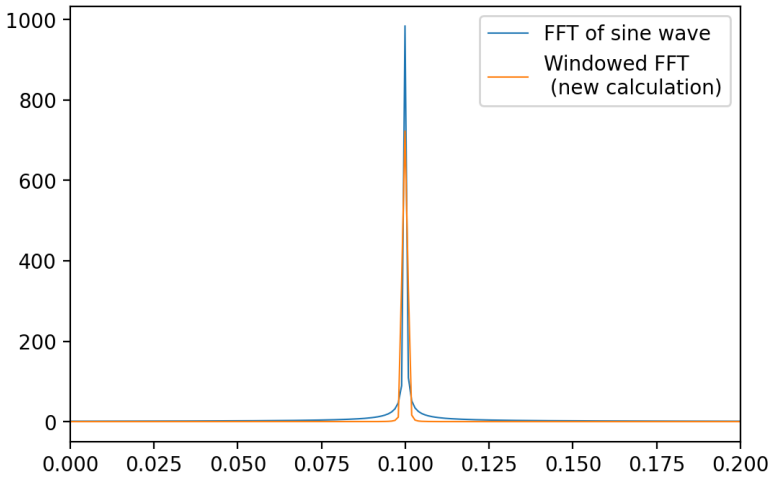
Since the convolution will be normalized to avoid power loss from the window function, we can remove the factors of N now instead of dividing by a much larger normalization factor later:

$$F[k]\ \rightarrow\ -F[k-1]/4 + F[k]/2 - F[k+1]/4$$

Implementing this into code and plotting:

```
ft_wndw_new=np.empty(N)
for i in range(N):
    if i==N-1:  # For wrap around, use special case for last array element
        ft_wndw_new[i]=-ft[i-1]/4+ft[i]/2+-ft[0]/4
    else:
        ft_wndw_new[i]=-ft[i-1]/4+ft[i]/2+-ft[i+1]/4

plt.plot(kshift,ftshift)
norm_new=np.sqrt(np.mean(ft_wndw_new**2))
ft_wndw_new=np.abs(ft_wndw_new)/norm
plt.plot(kshift,np.fft.fftshift(ft_wndw_new))
plt.xlim([0,0.2])
```

Plot on next page:



As expected, this is the same as the previous plot.

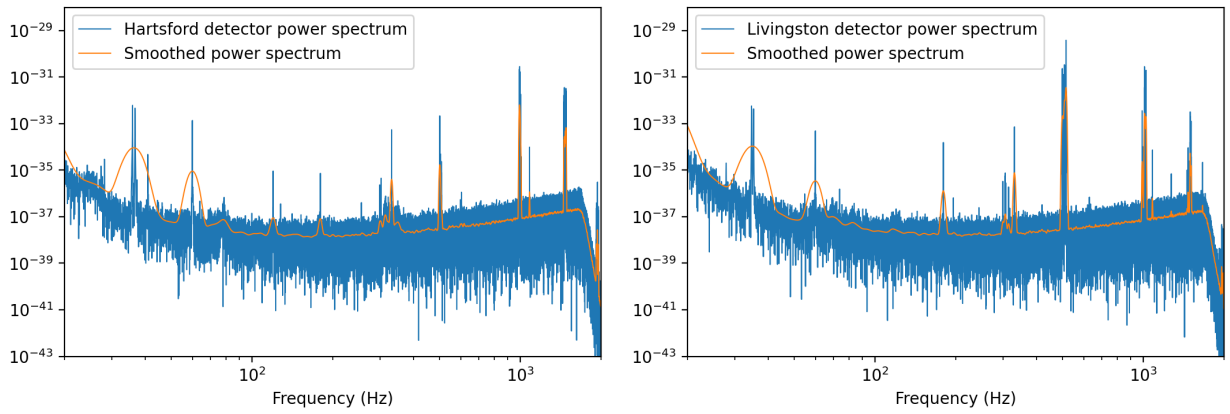
5 Question 5

a)

In order to estimate the noise for each detector, their power spectra had to first be calculated and smoothed. To do this, the strain data was first multiplied by a window function. A Tukey window from `scipy.signal` was used in this case which has an extended flat top in the middle. To then smooth the power spectrum, it was then convolved using a Gaussian kernel:

```
from A6Q1 import convolve,gaussian

def get_powerspect(time,strain,win,smooth_width):
    dt=time[1]-time[0]
    freqs=np.fft.fftfreq(len(time),dt)
    ps=np.abs(np.fft.fft(strain*win))**2
    freqs=np.fft.fftshift(freqs)  # FFT shift to convolve and plot properly
    ps=np.fft.fftshift(ps)
    smooth_fun=gaussian(freqs,0,smooth_width)
    ps_smooth=convolve(ps,smooth_fun/smooth_fun.sum())  # Normalize smooth_fun
    return freqs,ps,ps_smooth
```

Here are the plots for both the power spectrum and the smoothed power spectrum for event GW150914:



We then take the noise to be the square root of the smoothed power spectrum, therefore the matrix N^{-1} is given by 1 over the smoothed power spectrum (assuming Gaussian, stationary noise).

b) & c)

To search these events for gravitational waves using these noise models, a few functions were first written, the first of which calculates the matched filter of some windowed data and a give gravitational wave template:

```
def matched_filter(strain,win,template,noise_model):
    data_ft=np.fft.fft(win*strain)
    template_ft=np.fft.fft(win*template)
    Ninv_ft=1/noise_model
    rhs=np.fft.ifft(np.conj(template_ft)*Ninv_ft*data_ft)
    return np.fft.fftshift(rhs)  # FFT shift after inverse
```

From a matched filter, we can calculate the noise to signal ratio. The noise will be approximated as the RMS of the matched filter.

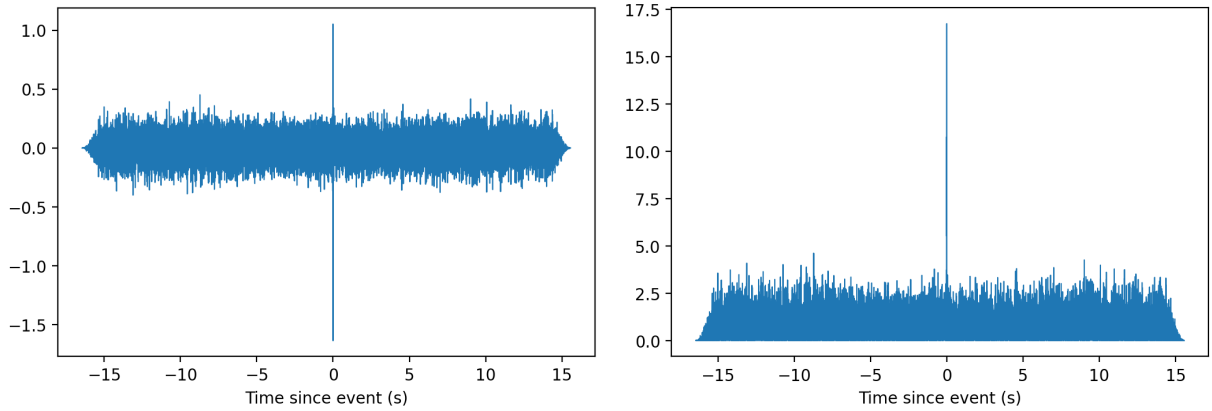
```
def signal2noise(matched_filter,win,template,noise_model,df):
    noise=np.sqrt(np.mean(matched_filter**2))  # Estimate noise as RMS of matched filter
    return np.abs(matched_filter)/noise
```

Now the data from each point even needs to be read and a matched filter passed over it. Here are two more functions that perform this for all the events:

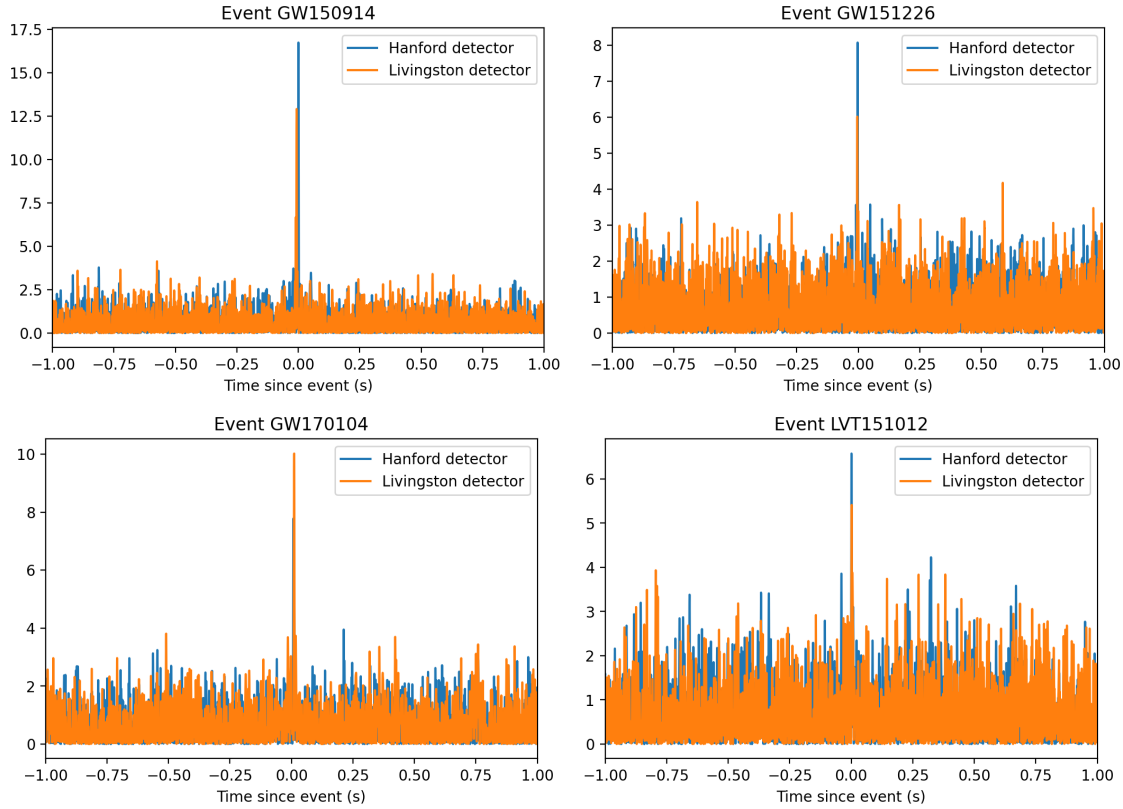
```
def read_template(data_dir,filename):
    dataFile=h5py.File(data_dir+filename,'r')
    template=dataFile['template']
    tp=template[0]
    tx=template[1]
    return tp,tx

def search_allevents(json_fname,event_names):
    events=json.load(open(data_dir+json_fname,'r'))
    time={}
    ps={}
    ps_smooth={}
    matched_filters={}
    signals2noise={}
    for event_name in event_names:
        event=events[event_name]   # First we have to read from JSON file
        fn_H1=event['fn_H1']
        fn_L1=event['fn_L1']
        tevent=event['tevent']
        strain_Ha,time_Ha,chan_dict_Ha=readligo.loaddata(data_dir+fn_H1,'H1')
        strain_Li,time_Li,chan_dict_Li=readligo.loaddata(data_dir+fn_L1,'L1')
        t=time_Ha-tevent
        template=read_template(data_dir,event['fn_template'])[0]
        win=scipy.signal.tukey(len(time),0.1)

        # Get noise model, then matched filter, then signal to noise
        freqs,ps_Ha,ps_smooth_Ha=get_powerspect(time,strain_Ha,win,2)
        noise_model_Ha=np.fft.fftshift(ps_smooth_Ha)
        matched_filter_Ha=matched_filter(strain_Ha,win,template,noise_model_Ha)
        df=np.abs(freqs[1]-freqs[0])
        signal2noise_Ha=signal2noise(matched_filter_Ha,win,template,noise_model_Ha,df)

        freqs,ps_Li,ps_smooth_Li=get_powerspect(time,strain_Li,win,2)
        noise_model_Li=np.fft.fftshift(ps_smooth_Li)
        matched_filter_Li=matched_filter(strain_Li,win,template,noise_model_Li)
        signal2noise_Li=signal2noise(matched_filter_Li,win,template,noise_model_Li,df)

        time[event_name]=t
        ps[event_name]=ps_Ha,ps_Li
        ps_smooth[event_name]=ps_smooth_Ha,ps_smooth_Li
        matched_filters[event_name]=matched_filter_Ha,matched_filter_Li
```

```
        signals2noise[event_name]=signal2noise_Ha,signal2noise_Li
    return time,freqs,ps,ps_smooth,matched_filters,signals2noise
```

Here is an example of the matched filter and its signal to noise ratio for the Hartsford detector during event GW150914:



Here are the plots of the signal to noise ratios from the matched filters for all the events:



f)

The difference in time of arrival for each event was calculated from the difference in the times for the peak SNR values at the 2 detectors. These were found to be (in seconds):

[0.00732421875, 0.001220703125, 0.003173828125, 0.00048828125]

for the events GW150914, GW151226, GW170104 and LVT151012 respectively. Multiplying by the speed of light gives a distance (in km):

[2197.265625, 366.2109375, 952.1484375, 146.484375]

This is what we expect: the two detectors are a few thousand kilometers apart. The maximum time separation occurs when the detectors and the wave source are all place along the same axis and when multiplied by the speed of light, it gives the spatial separation of the detectors. When the wave source is not on the same axis as both detectors, this time separation will be smaller. This explains the range of values we see above, the maximum of which is roughly the separation of the detectors.