Assignment 6

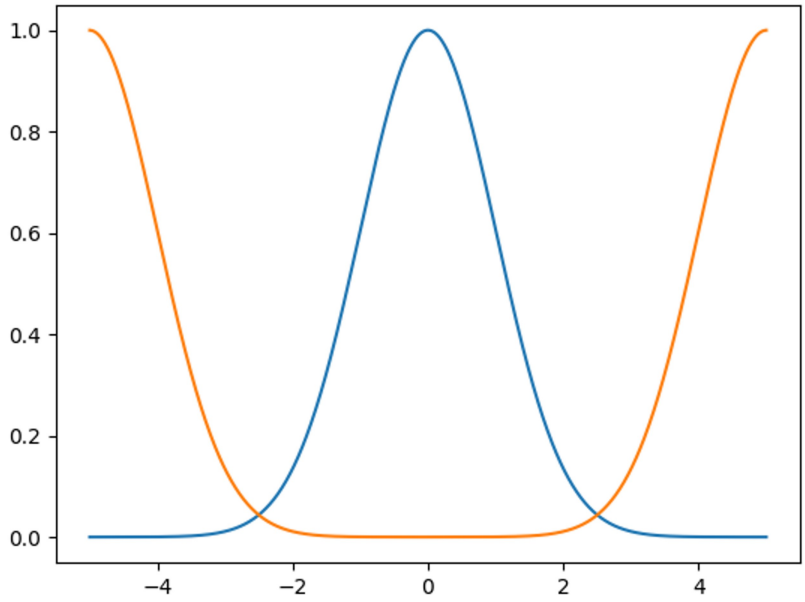
1.

Recall from class that a shift can be done by

$F(k^*) = \exp(2\pi i k dx/N) \cdot F(k)$

so we can simply inverse FT the product of $\exp(2\pi i k dx/N)$ and FT of the array to get the shifted function.

```
import numpy as np
from matplotlib import pyplot as plt

def shift_fun(y,dx):
    N=len(y)
    k=np.arange(len(y))
    ramp=np.exp(2*np.pi*1J*k*dx/N)
    yft=np.fft.fft(y)
    yft_shift=yft*ramp
    y_shift=np.fft.ifft(yft_shift)
    return y_shift

N=1000
x=np.linspace(-5,5,N)
y=np.exp(-0.5*x**2)
dx=N/2
y_shift=shift_fun(y,dx)

plt.plot(x,y)
plt.plot(x,y_shift)
plt.show()
```



2. $f * g = = \int f(x)\, g(x+y)\, dx$

$= \Sigma_x \sum F(k) \exp(2\pi i k x) \Sigma\, G(k') \cdot \exp(2\pi i k' y/N)$

$\cdot \exp(2\pi i k' x)$

$= \Sigma \sum F(k)\, G(k') \exp(2\pi i k' y/N) \underbrace{\Sigma_x \exp(2\pi i (k+k') x)}$

0 unless $k = -k'$

$= \Sigma F(-k') G(k') \exp(2\pi i k' y/N)$

$= \text{ift}(\ \text{conj}(\text{dft}(f)) \cdot \text{dft}(g)\ )$

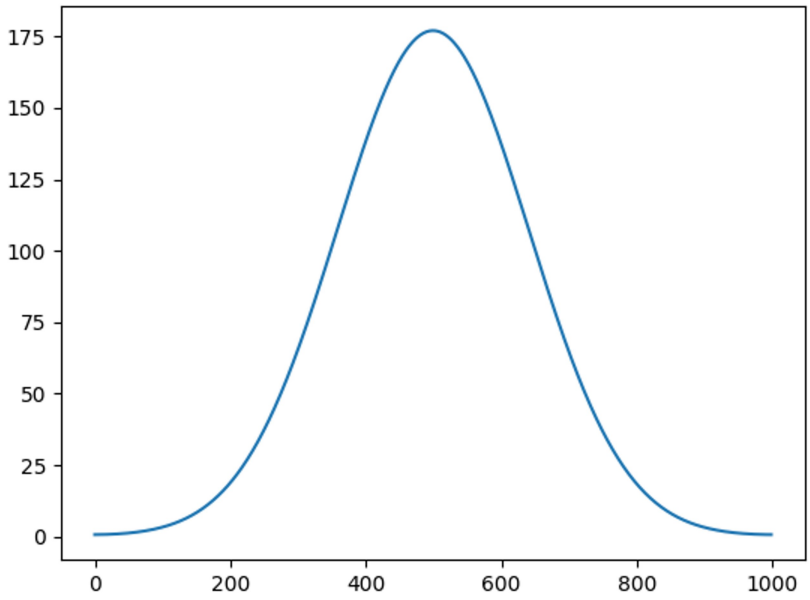
$= \text{ift}(\ \text{dft}(f) \cdot \text{conj}(\text{dft}(g))\ )$ if $f*g$ is real

$= \text{ift}(\text{dft}(f) \cdot \text{conj}(\text{dft}(g)))$ if $f * g$ is real

We then can use a similar code as convolution. My plot is aliased. I used Q1's code for shifting.

```
import numpy as np
from matplotlib import pyplot as plt

def correlation(f,g):
    F=np.fft.fft(f)
    G=np.fft.fft(g)
    H=F*np.conj(G)
    h=np.fft.ifft(H)
    return h

def shift_fun(y,dx):
    N=len(y)
    k=np.arange(len(y))
    ramp=np.exp(2*np.pi*1J*k*dx/N)
    yft=np.fft.fft(y)
    yft_shift=yft*ramp
    y_shift=np.fft.ifft(yft_shift)
    return y_shift

N=1000
x=np.linspace(-5,5,N)
y=np.exp(-0.5*x**2)

def corr_gaussian(dx):
    y_shift = shift_fun(y,dx)
    cor = correlation(y,y_shift)
    return cor

for i in range(0,500,50):
    h=corr_gaussian(i)
    cut = int(N/2)
    h = np.concatenate((h[cut+1:],h[:cut]))
    plt.plot(h)
    plt.savefig(f"corr_gaussian_{i}.png")
    plt.show()
```

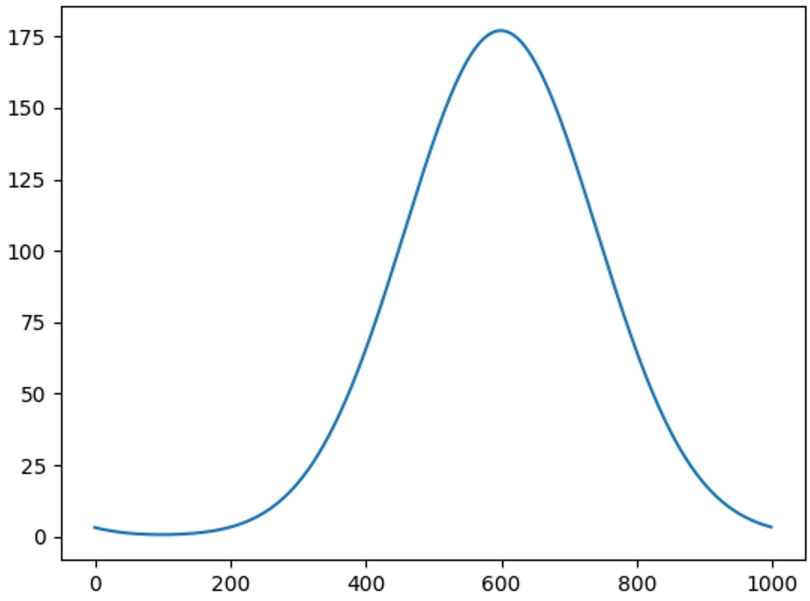
for $i=0$, we have the correlation of a gaussian with itself.



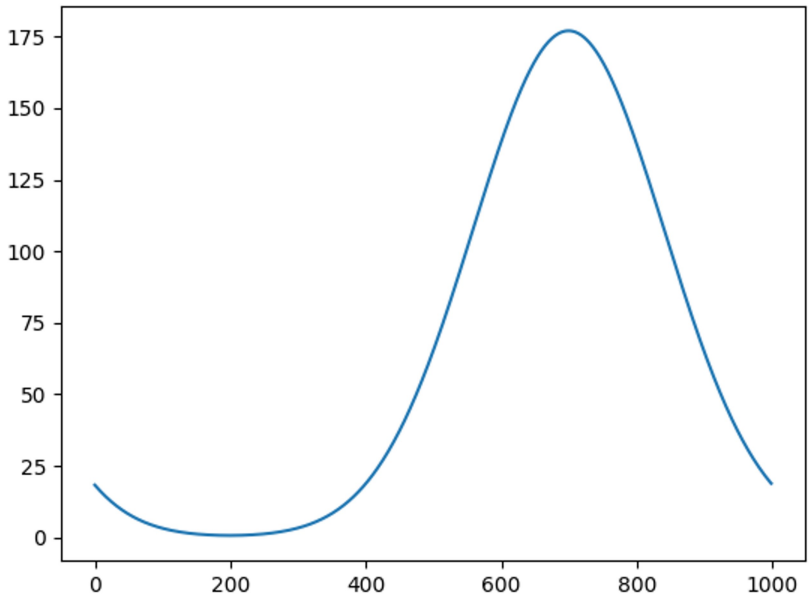
It is still a gaussian.

As we increase the shift in gaussian;

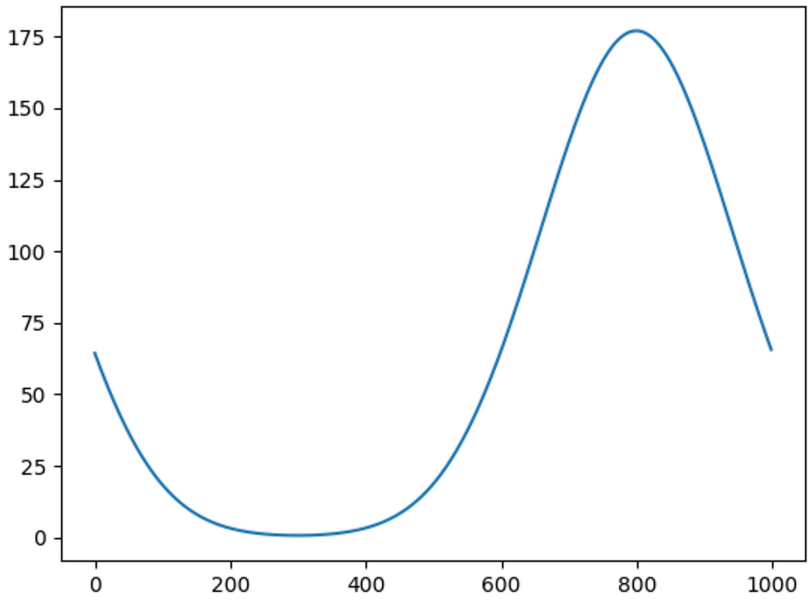
$dx = 100$



dx = 200

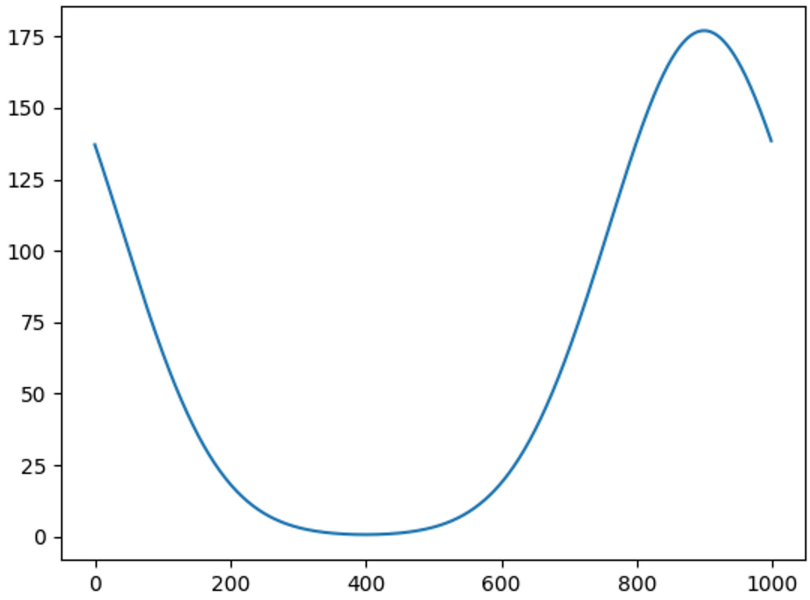


dx = 300



dx = 400

dx = 400



we see that our correlation also shifted to the right as we shift our gaussian to the right by the same amount.

which is to be expected.

3. we can add zeros to the end of arrays such that the length of the array is $2N-1$, where N is the original length.

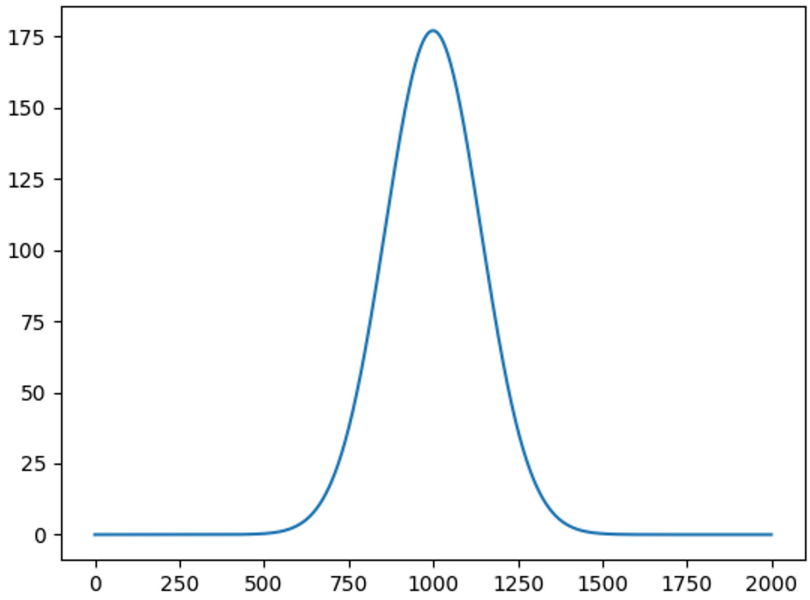
This is called zero-padding.

I did a little search, the reason

I did a little search, the reason why this works is that it makes the circular convolution have the same length of linear convolution. I am not sure how the math works out here, if there is a reference, please let me know.

```
import numpy as np
from matplotlib import pyplot as plt

def convolution(f,g):
    f_new = add_zeros(f)
    g_new = add_zeros(g)
    F=np.fft.fft(f_new)
    G=np.fft.fft(g_new)
    H=F*G
    h=np.fft.ifft(H)
    return h

def add_zeros(array):
    N=len(array)
    zeros = np.zeros(N-1)
    new_array = np.concatenate((array,zeros))
    # cut = int(len(new_array)/2)
    # new_array = np.concatenate((new_array[cut+1:],new_array[:cut]))
    return new_array

N=1000
cut=int(N/2)
x=np.linspace(-5,5,N)
y1=np.exp(-0.5*x**2)
arr = convolution(y1,y1)

plt.plot(arr)
plt.show()
```

for a test, we convolute two identical gaussian,



we can see that the result is also gaussian, as expected.

4.

(a)

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

$$= \sum_{x=0}^{N-1} \left[\exp(-2\pi i k/N)\right]^x$$

Note $\sum_{x=0}^{N-1} a^x = \frac{1-a^N}{1-a}$ (geometric series)

$$= \frac{1-\exp(-2\pi i k)}{1-\exp(-2\pi i k/N)}$$

(b) use l'hopital's rule

$$\lim_{k \to 0} \frac{1-\exp(-2\pi i k)}{1-\exp(-2\pi i k/N)}$$

$$= \lim_{k \to 0} \frac{2\pi i k \exp(-2\pi i k)}{2\pi i k/N \cdot \exp(-2\pi i k/N)}$$

$$= \frac{1}{\frac{1}{N}} = N$$

for integer $k \neq$ multiple of N,

top $= 0$, bottom $\neq 0$, so

it is zero

If it is multiple of N, it

reduces to the case where $k=0$.

c. a sine wave with k_0 full sine wave

can be written as

$\quad {}^{ik_0 x} \qquad {}^{-2\pi i k_0 x}$

can be written as

$$\cos\left(\frac{2\pi i k_0 x}{N}\right) = \frac{e^{\frac{2\pi i k_0 x}{N}} - e^{\frac{-2\pi i k_0 x}{N}}}{2i}$$

$$DFT(\cos) = \sum_{x=0}^{N-1} e^{\frac{-2\pi i k x}{N}} \cdot \frac{e^{\frac{2\pi i k_0 x}{N}} - e^{\frac{-2\pi i k_0 x}{N}}}{2i}$$

$$= \frac{1}{2i} \sum_{x=0}^{N-1} e^{\frac{-2\pi i (k-k_0) x}{N}} - e^{\frac{-2\pi i (k+k_0) x}{N}}$$

$$= \frac{1}{2i} \cdot \left[\frac{1 - e^{-2\pi i (k-k_0)}}{1 - e^{-2\pi i (k-k_0)/N}} - \frac{1 - e^{-2\pi i (k+k_0)}}{1 - e^{-2\pi i (k+k_0)/N}} \right]$$

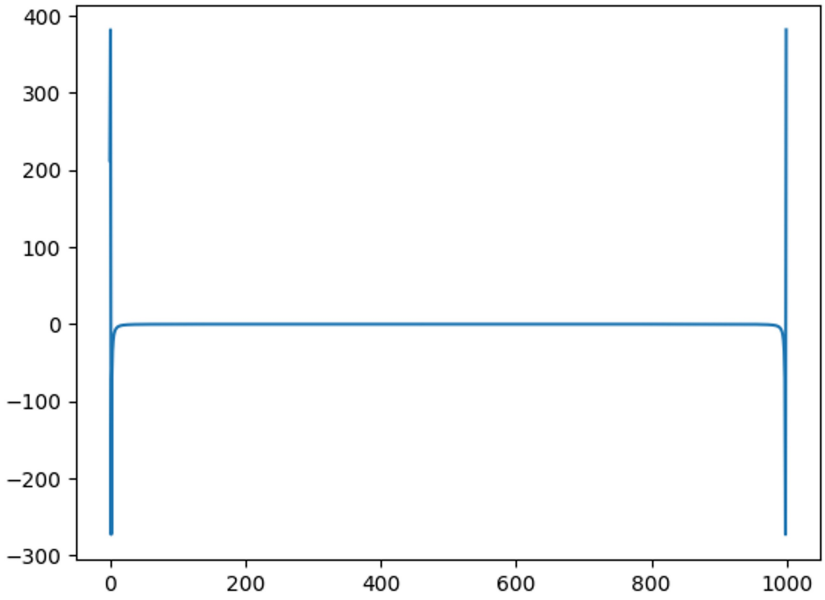
we can then pick a none integer k_0

In the following code we used $k_0 = 1.5$

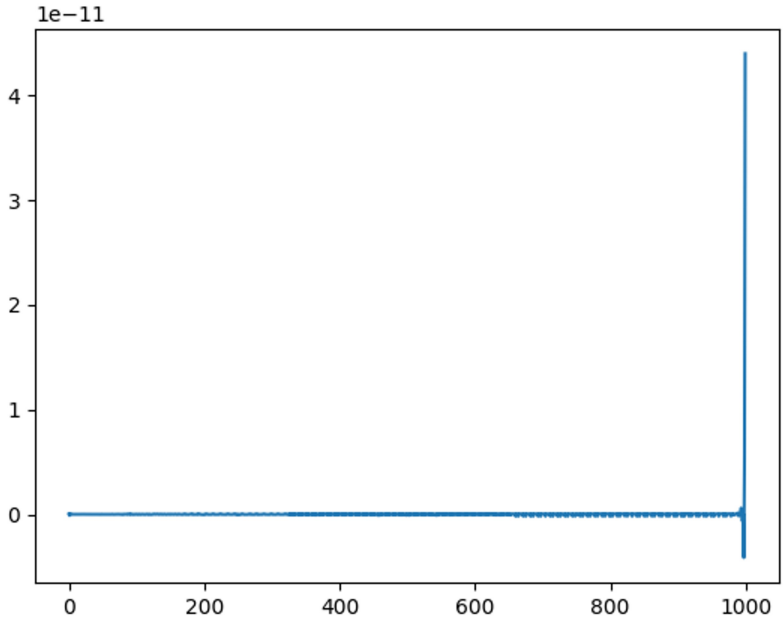
```
import numpy as np
from matplotlib import pyplot as plt


N=1000
cut=int(N/2)
x = np.asarray(range(N))

k0=1.5
y1=np.sin(2*np.pi*k0*x/N)
k = np.asarray(range(N))

dft_ana = 1/(2J)*((1-np.exp(-2*np.pi*1J*(k-k0)))/(1-np.exp(-2*np.pi*1J*(k-k0)/N))
-(1-np.exp(-2*np.pi*1J*(k+k0)))/(1-np.exp(-2*np.pi*1J*(k+k0)/N)))

dft_fft = np.fft.fft(y1)

plt.plot(dft_ana)
```

```
plt.show()
plt.plot(dft_ana-dft_fft)
plt.show()
```

The dft we get is:



The difference between analytic & fft is:

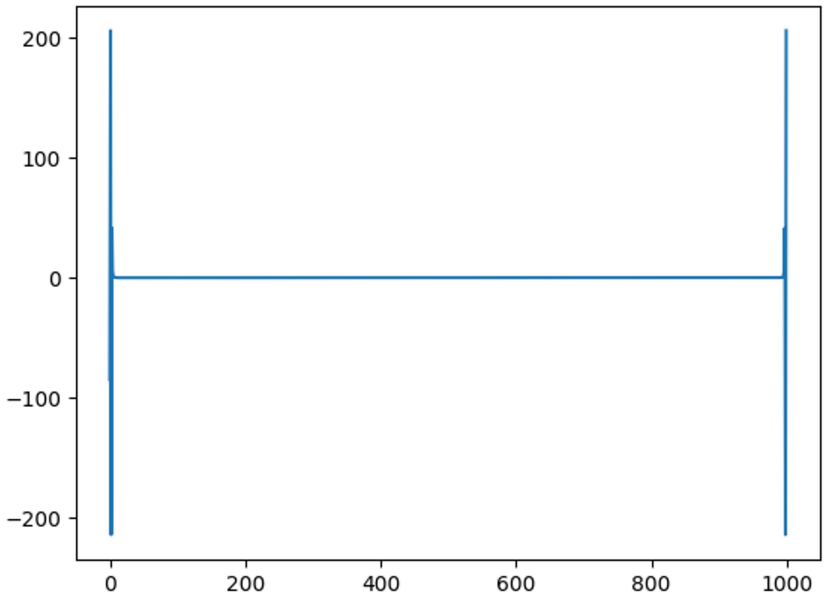


we can see that analytic & fft agrees very well.

The sine function DFT should be imaginary delta functions, so it should be zero in real space. But here we are not getting that.

d.

```
win = 0.5 - 0.5*np.cos(2*np.pi*x/N)
win_y = win*y1
plt.plot(np.fft.fft(win_y))
plt.show()
```



we can see that we reduces the leakage by about a factor of 2.

(e)

we can use numerical:

```
print(np.fft.fft(win))
```

output is:

```
[ 5.00000000e+02+0.00000000e+00j -2.50000000e+02+2.63677968e-14j
 -7.65399890e-15+8.03530688e-16j -1.33534904e-15-1.90553443e-15j
  1.83389915e-15+1.49690855e-15j  1.64387448e-15+5.71239744e-16j
...
  1.83389915e-15-1.49690855e-15j  1.46327125e-15+2.42717908e-15j
 -7.65399890e-15-8.03530688e-16j -2.50000000e+02-3.30168710e-14j]
```

so we see that it is indeed

$$\left[\frac{N}{2} \quad -\frac{N}{2} \quad 0 \ldots \quad 0 \quad -\frac{N}{2}\right]$$

$$\left[\frac{N}{2}, -\frac{N}{4}, 0, \ldots, 0, -\frac{N}{4}\right]$$

Analytical derivation can be found in Wikipedia of Hann function.

we multiply in real space
→ convolution in fourier space.

$$DFT(f \cdot win)(k')$$

$$= \sum_k DFT(f)(k) \cdot DFT(win)(k-k')$$

Note $DFT(win)(k-k')$

$$= \begin{cases} \frac{N}{2} & \text{if } k = k' \\ -\frac{N}{4} & \text{if } k = k' \pm 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\Rightarrow DFT(f \cdot win)(k)$$

$$= \frac{N}{2} DFT(f)(k) - \frac{N}{4} DFT(f)(k+1) - \frac{N}{4} DFT(f)(k-1)$$

5.

First we set up our reading file codes.

```
import numpy as np
from matplotlib import pyplot as plt
import h5py
import glob

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

fname_h = []
fname_l = []
fname_temp = []
event_name = []

fname_h.append('H-H1_LOSC_4_V2-1126259446-32.hdf5')
fname_l.append('L-L1_LOSC_4_V2-1126259446-32.hdf5')
fname_temp.append('GW150914_4_template.hdf5')
event_name.append('GW150914')

fname_h.append('H-H1_LOSC_4_V2-1128678884-32.hdf5')
fname_l.append('L-L1_LOSC_4_V2-1128678884-32.hdf5')
fname_temp.append('LVT151012_4_template.hdf5')
event_name.append('LVT151012')
```

```
fname_h.append('H-H1_LOSC_4_V2-1135136334-32.hdf5')
fname_l.append('L-L1_LOSC_4_V2-1135136334-32.hdf5')
fname_temp.append('GW151226_4_template.hdf5')
event_name.append('GW151226')

fname_h.append('H-H1_LOSC_4_V1-1167559920-32.hdf5')
fname_l.append('L-L1_LOSC_4_V1-1167559920-32.hdf5')
fname_temp.append('GW170104_4_template.hdf5')
event_name.append('GW170104')
```

we steal most of the code from simple_read_ligo.py from class repo.

we set up file names in a list so that we can choose which event to analyse.

In the following section, I will comment what I did and show my code for each small question, but I will show the results in the very end.

(a)

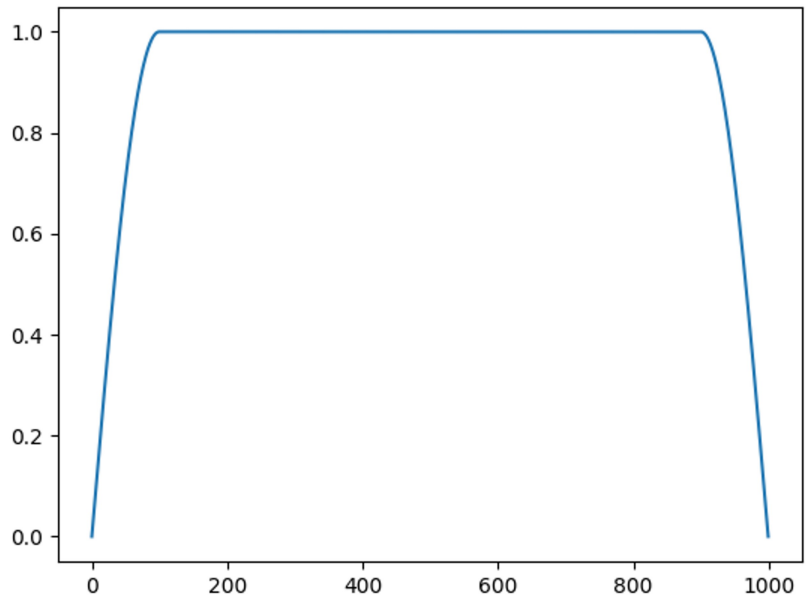
we smooth the power spectrum the same way as we did in class, which is to simply convolve the data with a Gaussian kernal.

As for window function, I discussed with Sam, and decided to use a

I discussed with [illegible]
and decided to use a
flattened cosine function,
which is a cosine function,
but with middle part being
ones.

Also, we cutted the frequency
below 20 and above 1500,
as what we did in class. We
set these to zero.

The window function looks
like this




```
def smooth_vector(vec,sig):
    n=len(vec)
    x=np.arange(0,n,1.0)
```

```
    x[n//2:]=x[n//2:]-n
    kernel=np.exp(-0.5*((x/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 window_fun(n):
    y=np.ones(n)
    cos_n = n//5
    cos_x = np.linspace(-np.pi/2,np.pi/2,cos_n)
    cos_y = np.cos(cos_x)
    cut = cos_n//2
    y[:cut]=cos_y[:cut]
    y[-cut:]=cos_y[-cut:]
    return y

def ligo_analysis(n):
    hname=fname_h[n]
    lname=fname_l[n]
    strain_h,dt_h,utc_h=read_file(hname)
    strain_l,dt_l,utc_l=read_file(lname)
    ename = event_name[n]
    template_name=fname_temp[n]
    tp,tx=read_template(template_name)
    length = len(strain_h)

    win = window_fun(length)

    noise_ft_h=np.fft.fft(win*strain_h)
    noise_smooth_h=smooth_vector(np.abs(noise_ft_h)**2,10)
    noise_smooth_h=noise_smooth_h[:len(noise_ft_h)//2+1] #will give us same
length

    noise_ft_l=np.fft.fft(win*strain_l)
    noise_smooth_l=smooth_vector(np.abs(noise_ft_l)**2,10)
    noise_smooth_l=noise_smooth_l[:len(noise_ft_l)//2+1] #will give us same
length

    tobs=dt_h*length
    dnu=1/tobs
    nu=np.arange(len(noise_smooth_h))*dnu
    nu[0]=0.5*nu[1]

    Ninv_h=1/noise_smooth_h
    Ninv_h[nu>1500]=0
    Ninv_h[nu<20]=0

    Ninv_l=1/noise_smooth_l
    Ninv_l[nu>1500]=0
    Ninv_l[nu<20]=0
```

starting from here, all codes in the following is part of

in the following is part of
the function ligo.analysis.

(b)
we did a matched filter
using whitened noise & template.

$$m = \frac{AN^{-\frac{1}{2}} \circledast dN^{-\frac{1}{2}}}{(AN^{-\frac{1}{2}}) \cdot (AN^{-\frac{1}{2}})}$$

$$top = IFT\left(FT(dN^{-\frac{1}{2}}) \cdot conj(FT(AN^{-\frac{1}{2}})) \right)$$

Most of the following code
is copied from the
mf_ligo.class.py from class
repo.

```
template_ft_white_h=np.fft.rfft(tp*win)*np.sqrt(Ninv_h)
data_ft_white_h=np.fft.rfft(strain_h*win)*np.sqrt(Ninv_h)
template_white_h = np.fft.irfft(template_ft_white_h)
data_white_h = np.fft.irfft(data_ft_white_h)
rhs_h = np.fft.irfft(data_ft_white_h*np.conj(template_ft_white_h))
lhs_h = np.dot(template_white_h,template_white_h)
m_h = rhs_h/lhs_h
m_h_shifted = np.fft.fftshift(m_h)

template_ft_white_l=np.fft.rfft(tp*win)*np.sqrt(Ninv_l)
data_ft_white_l=np.fft.rfft(strain_l*win)*np.sqrt(Ninv_l)
template_white_l = np.fft.irfft(template_ft_white_l)
data_white_l = np.fft.irfft(data_ft_white_l)
rhs_l = np.fft.irfft(data_ft_white_l*np.conj(template_ft_white_l))
lhs_l = np.sum(template_white_l**2)
m_l = rhs_l/lhs_l
m_l_shifted = np.fft.fftshift(m_l)
```

(c) & (d)

$$SNR = \frac{\text{signal}}{\text{noise}}$$

we can get signal height by naively taking the maximum of our matched filter result.

absolute value of

Numerical noise is simply the standard deviation of our matched filter.

Analytic noise is $\frac{1}{\sqrt{\text{length} \cdot \text{lhs}}}$. The reason of the $\frac{1}{\sqrt{\text{length}}}$ normalization can be found on last page of class PPT.

From this post, the combined SNR is

https://dsp.stackexchange.com/questions/58728/combining-snr-measurements

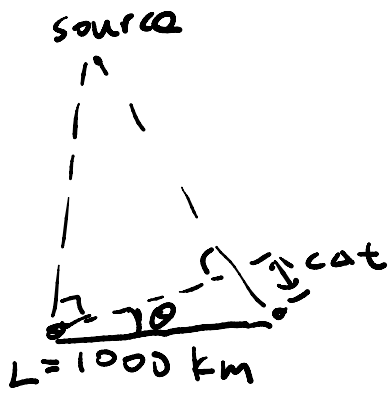
$$\frac{1}{2}\left(\sqrt{SNR_1} + \sqrt{SNR_2}\right)^2 = SNR\_combined$$

```
signal_height_h = np.max(np.abs(m_h))
num_noise_h = np.std(m_h)
ana_noise_h = 1/np.sqrt(length*lhs_h)
num_SNR_h = signal_height_h/num_noise_h
ana_SNR_h = signal_height_h/ana_noise_h

signal_height_l = np.max(np.abs(m_l))
num_noise_l = np.std(m_l)
ana_noise_l = 1/np.sqrt(length*lhs_l)
num_SNR_l = signal_height_l/num_noise_l
ana_SNR_l = signal_height_l/ana_noise_l

num_SNR_total = 0.5*(np.sqrt(num_SNR_h)+ np.sqrt(num_SNR_l))**2
ana_SNR_total = 0.5*(np.sqrt(ana_SNR_h)+ np.sqrt(ana_SNR_l))**2
```

(e) I discussed with Sam, and he offered a way of doing this without using a cumulative sum. We could simply use the whitened template fourier transform as a weight to average the frequencies. Then the weighted average would be the frequency where half power comes from below & half comes above. This is the same as doing a cumulative sum, but easier to implement.

```
weight_h = np.abs(template_ft_white_h)**2
weight_l = np.abs(template_ft_white_l)**2
half_v_h = np.average(nu,weights = weight_h)
half_v_l = np.average(nu,weights = weight_l)
```

(f)



Discussed with Sam.
From slacks we get roughly

Discussed with Sam.

From slacks, we get roughly this diagram.

source is very far away,

$\Rightarrow \theta$ very small

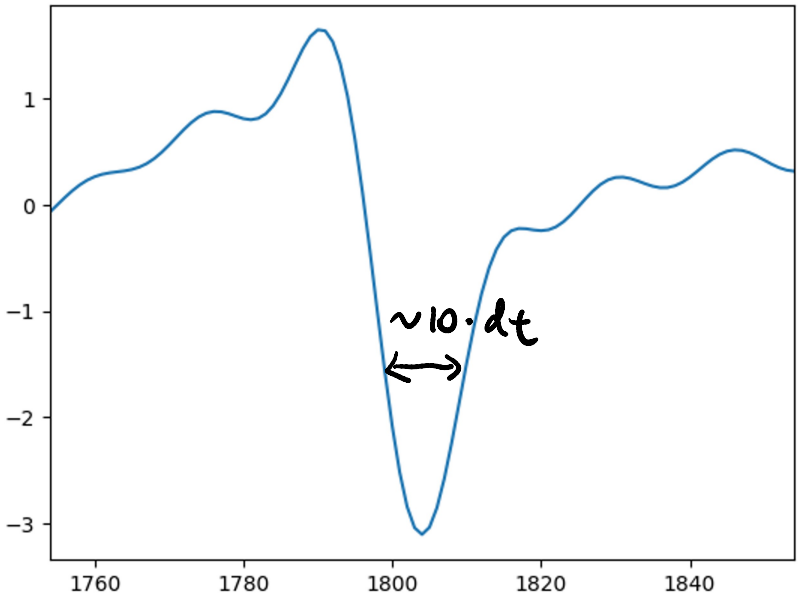
$\Rightarrow \quad \theta \approx \frac{c\Delta t}{L}$

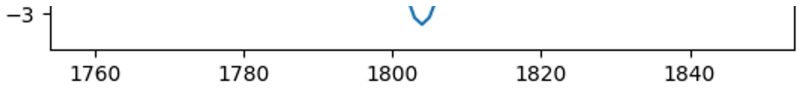
$\Rightarrow \sigma_\theta = \frac{c\,\sigma_{\Delta t}}{L}$

Δt is simply time between matched filter peaks of the two detectors.

we can sort of eye-ball $\sigma_{\Delta t}$ from a zoomed in mf result.

Hanford for GW150914





we can see that the peak in mf is roughly a gaussian with width of $10 \cdot dt$. I checked several peaks and this is quite universal.

$$\Delta t = t_{\text{hanford}} - t_{\text{livingston}}$$

$$\Rightarrow \sigma_{\Delta t} = \sqrt{\sigma_{t_{\text{hanford}}}^2 + \sigma_{t_{\text{livingston}}}^2}$$

$$= 10\sqrt{2}\, dt$$

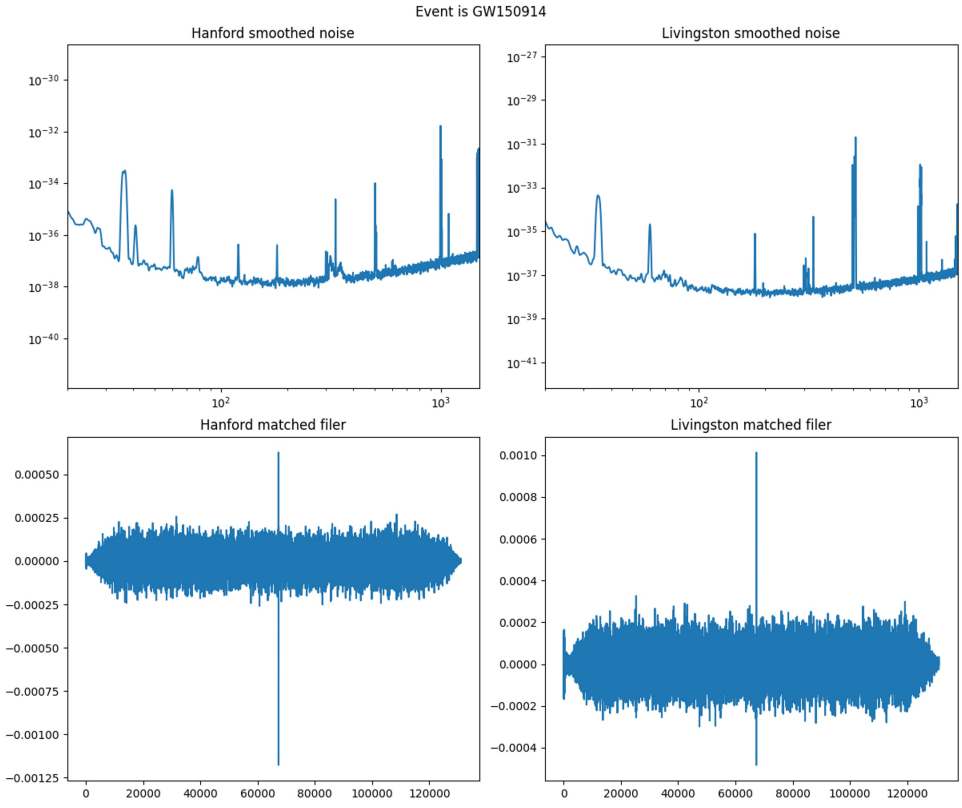
```
t_diff = np.abs(np.argmax(np.abs(m_h))-np.argmax(np.abs(m_l)))*dt_h
t_diff_var = np.sqrt(2)*10*dt_h
angle = 3e5*t_diff/1e3 + np.pi
angle_var = 3e5*t_diff_var/1e3
```

Data analysis.

```
fig = plt.figure(figsize = (12,10),constrained_layout = True)
axes = fig.subplots(2,2).flatten()
axes[0].loglog(nu,noise_smooth_h)
axes[0].set_title('Hanford smoothed noise')
axes[0].set_xlim(20,1500)
axes[1].loglog(nu,noise_smooth_l)
axes[1].set_title('Livingston smoothed noise')
axes[1].set_xlim(20,1500)
axes[2].plot(m_h_shifted)
axes[2].set_title('Hanford matched filer')
axes[3].plot(m_l_shifted)
axes[3].set_title('Livingston matched filer')
plt.suptitle(f'Event is {ename}')
plt.savefig(f'ana_{ename}.png')

print('\nEvent is ',ename)
```

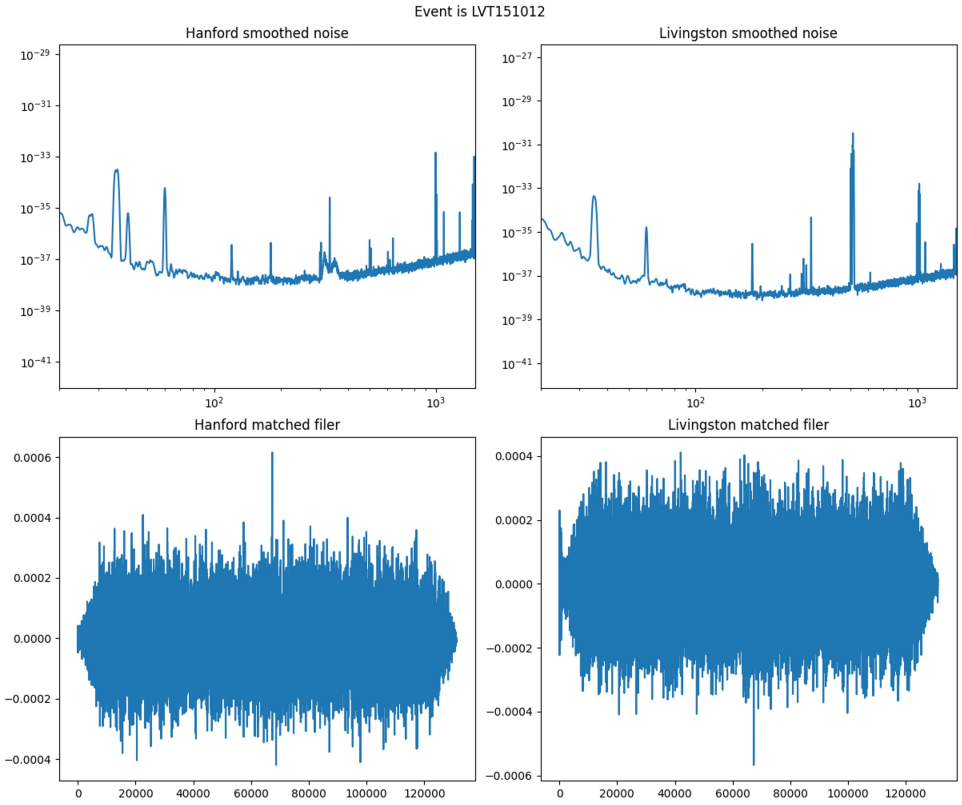
```
print('Hanford Numerical SNR =',num_SNR_h)
print('Hanford Analytic SNR =',ana_SNR_h)
print('Livingston Numerical SNR =',num_SNR_l)
print('Livingston Analytic SNR =',ana_SNR_l)
print('Total Numerical SNR =',num_SNR_total)
print('Total Analytic SNR =',ana_SNR_total)
print('Hanford Half-weight frequency is: ', half_v_h)
print('Livingston Half-weight frequency is: ', half_v_l)
print('Time difference is: ',t_diff,'+-',t_diff_var)
print('Angle of the source is: ',angle,'+-',angle_var)
```

GW150914



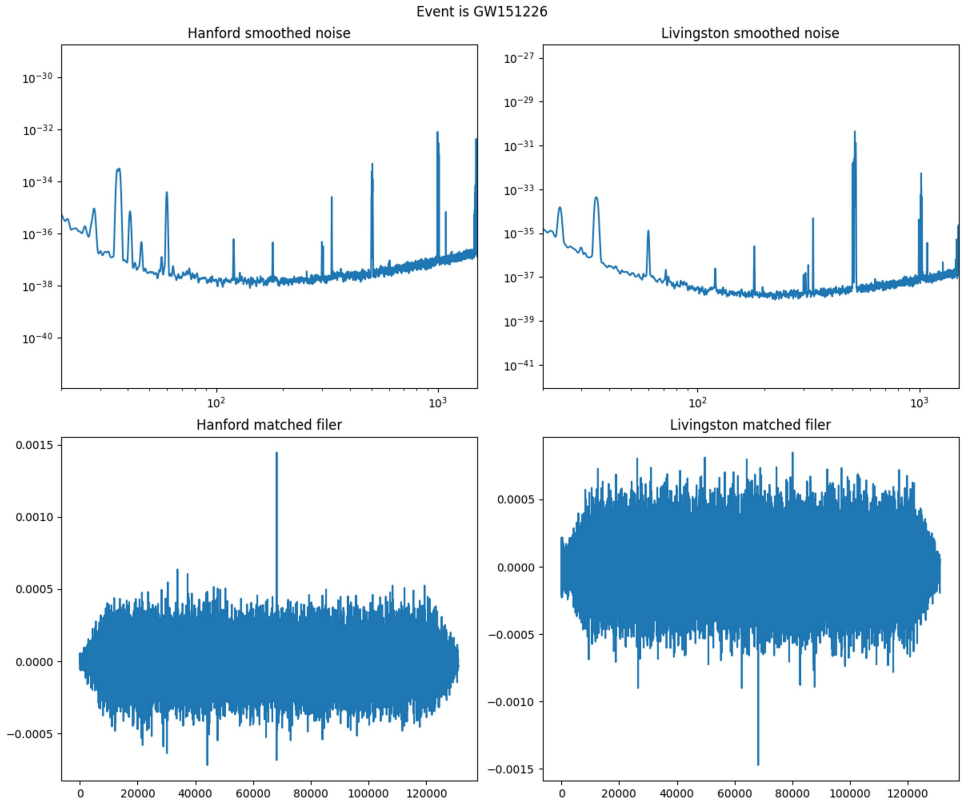
Event is GW150914
Hanford Numerical SNR = 18.606882629178692
Hanford Analytic SNR = 18.26867965595239
Livingston Numerical SNR = 13.910876470209999
Livingston Analytic SNR = 13.690600771605471
Total Numerical SNR = 32.34732402858045
Total Analytic SNR = 31.794481338304802
Hanford Half-weight frequency is: 117.28547495791555
Livingston Half-weight frequency is: 126.01256441700743
Time difference is: 0.00732421875 +- 0.003452669830012439
Angle of the source is: 5.338858278589793 +- 1.0358009490037319

LVT 151012



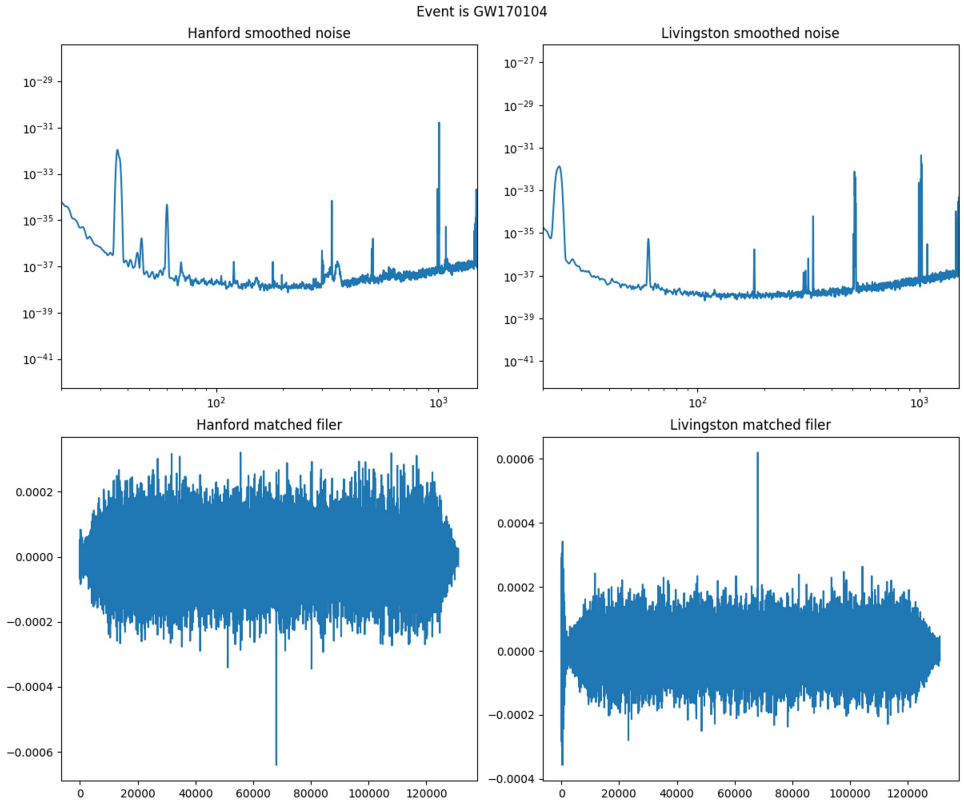
Event is LVT151012
Hanford Numerical SNR = 6.373811523676048
Hanford Analytic SNR = 6.245413057846358
Livingston Numerical SNR = 5.399782470243118
Livingston Analytic SNR = 5.309771822839578
Total Numerical SNR = 11.753414057428039
Total Analytic SNR = 11.53621365349347
Hanford Half-weight frequency is: 102.38315490827075
Livingston Half-weight frequency is: 116.43175991759179
Time difference is: 0.00048828125 +- 0.003452669830012439
Angle of the source is: 3.288077028589793 +- 1.0358009490037319

GW151226



Event is GW151226
Hanford Numerical SNR = 10.310542453859545
Hanford Analytic SNR = 10.066955154179047
Livingston Numerical SNR = 7.536738521154225
Livingston Analytic SNR = 7.414425975299084
Total Numerical SNR = 17.738846808745162
Total Analytic SNR = 17.38017514353867
Hanford Half-weight frequency is: 106.47204642852435
Livingston Half-weight frequency is: 143.5601287548851
Time difference is: 0.001220703125 +- 0.003452669830012439
Angle of the source is: 3.507803591089793 +- 1.0358009490037319

GW170104



Event is GW170104
Hanford Numerical SNR = 8.28472164456094
Hanford Analytic SNR = 8.115003149232933
Livingston Numerical SNR = 9.97580039750137
Livingston Analytic SNR = 9.827636686926617
Total Numerical SNR = 18.221285686875994
Total Analytic SNR = 17.9016784027055
Hanford Half-weight frequency is: 117.78016268296095
Livingston Half-weight frequency is: 101.01442824327403
Time difference is: 0.003173828125 +- 0.003452669830012439
Angle of the source is: 4.093741091089793 +- 1.0358009490037319