PHYS-512: Problem Set 6

Simon Briesenick

November 14, 2022

1. We need some kind of operator $\hat{T}_a : \mathbb{R} \to \mathbb{R}$ that maps an arbitrary, indexed input array $f(\tau)$ to $f(\tau + a)$, i.e.

$$\hat{T}_a f(\tau) = f(\tau + a) \tag{1}$$

for, the index τ between 0 and $N - 1$ and involves the convolution of the input array with another one, i.e.

$$h(a) = (f * g)(a) = \sum_{\tau=0}^{N-1} f(\tau) g(a - \tau) \tag{2}$$

such that $h(a) = f(\tau + a)$. If g is chosen as the discrete Dirac delta function, this is exactly the case. The convolution of two arrays can be understood as *smoothening and averaging* of one of the two arrays, say $f(\tau)$ with the other one. The discrete Dirac delta function centered at a will "sample" only a single entry of the input array with a weight of 1, which means that the resulting array will simply be a shifted copy of the input array. The result is shown in Fig. 1. The Code:

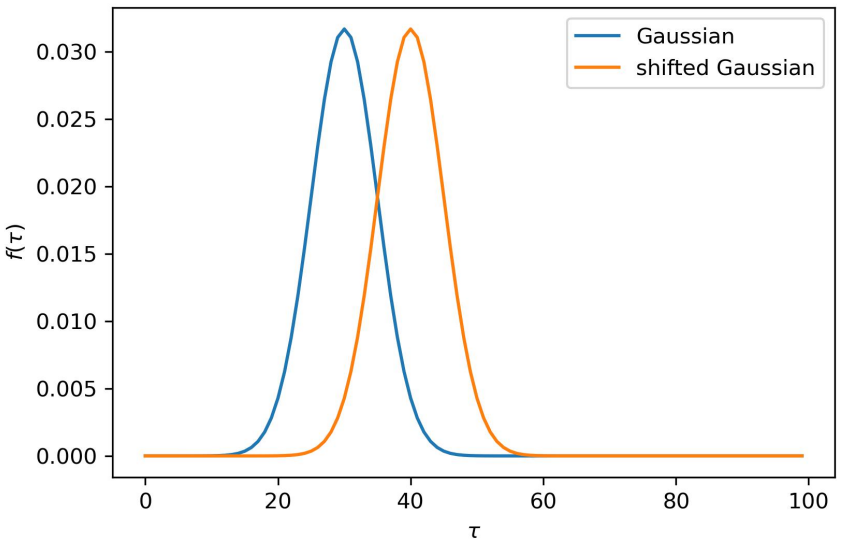

Figure 1: Shifted Gaussian by 10 units.

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

def fft_shift(arr, shift):
    # according to the convolution theorem, we can
    # write the convolution of two functions conv(f, g)
    # as the inverse fourier transrom of the product of
    # their respective fourier transforms, like
    # h(t) = conv(f, g)(t) = IFT(F*G)
    g = np.zeros(len(arr))
    g[shift] = 1
    arr_ft = np.fft.fft(arr)
    g_ft = np.fft.fft(g)
    return np.fft.ifft(arr_ft*g_ft)

x = np.arange(0, 101) # indices
sigma = 5
x0 = 30 # peak index
y = (1/sigma*np.sqrt(2*np.pi))**5*np.exp(-0.5*((x - x0)/sigma)**2)
shift = 10
y_shift = fft_shift(y, shift)
```

2. For the same Gaussian as in the previous question, the correlation with itself as well as shifted versions of itself is shown in Fig. 2. The auto-correlation of a shifted Gaussian comes out to be the

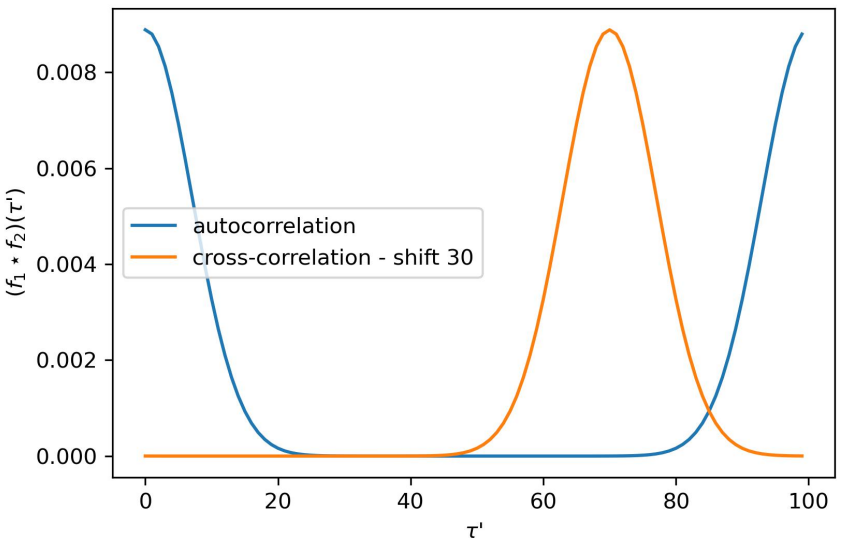

Figure 2: Gaussian correlated with itself (i.e. $f_1 = f_2$) and shifted version of itself (i.e. $f_2(\tau) = f_1(\tau + a)$).

same as that of an unshifted Gaussian, i.e. it does not depend on the shift. This is not surprising

because the auto-correlation

$$(f \star f)(\tau') = \sum_{\tau=0}^{N-1} f(\tau) f(\tau + \tau') \tag{3}$$

is independent on the ordering of τ, i.e. shifting the input array by some amount a as above would only amount to a change in what order the sum is computed, but not the terms itself. Picturing the array over a circle makes this aspect clear - Shifting a function is the same as rotating the ring which implies that the sum of the products of these values will be the same. The cross-correlation of an array with a shifted version of itself however does depend on the shift as seen on the right side in Fig. 2. In signal processing, the auto-correlation is a measure for the coherence of a signal. So it makes sense that it is maximized when $\tau' = 0$ and then decays to over an interval that is proportional to the σ of the input Gaussian. The second increase at the end of the index array is due to the cyclic nature of the DFT, i.e. because $f(\tau + N) = f(\tau)$. The cross-correlation peaks when the two displaced (but equal) signals lie on top of each other, which occurs when $\tau' + a = N$, so at $\tau' = 70$ for the case below (with $N = 100$).

3. The convolution can, like the correlation, in DFTs can be understood as the product of two arrays defined on the circle that are shifted with respect to each other by some offset (eq. 2). Wrapping around occurs when non-zero elements exist in the end and or the beginning of the two arrays which lead to non-zero elements in the sum $f(\tau)g(\tau + a)$. If we pretend the two arrays are actually double the size of the original ones, but with zeros for the second half of their length, the product will evaluate to 0 for all of the wrapped terms.

```python
import numpy as np

def no_wrapping(arr1, arr2):
    # assuming that len(arr1) = len(arr2) = N,
    # one can append N zeros to both arrays to
    # get rid of wrapping-around effects.
    N = len(arr1)
    zeros = np.zeros(N)
    a1 = np.append(arr1, zeros)
    a2 = np.append(arr2, zeros)
    dft_a1 = np.fft.fft(a1)
    dft_a2 = np.fft.fft(a2)

    # calculate convolution and ignore second
    # half of values
    conv = 1/N*np.fft.ifft(dft_a1*dft_a2)[:N]
    return conv
```

4. (a) The finite sum of the geometric series

$$\sum_{\tau=0}^{N-1} e^{-2\pi i(v/N)\tau} \tag{4}$$

can be rewritten as

$$\sum_{\tau=0}^{N-1} \left(e^{-2\pi i(v/N)}\right)^{\tau} = \sum_{\tau=0}^{N-1} r^{\tau}. \tag{5}$$

I take it as a given that I can use the fact that

$$\sum_{\tau=0}^{N-1} r^{\tau} = \begin{cases} \frac{1-r^N}{1-r} & r \neq 1 \\ N & r = 1. \end{cases} \tag{6}$$

Which means that for $r \neq 1$ eqn. 5 results in

$$\frac{1 - \exp\left(-2\pi i (v/N)\right)^N}{1 - \exp\left(-2\pi i (v/N)\right)} = \frac{1 - \exp\left(-2\pi i v\right)}{1 - \exp\left(-2\pi i (v/N)\right)}. \tag{7}$$

and N for $r = 1$.

(b) The complex exponential can be approximated with the MacLaurin series as

$$e^z = \sum_{n=0}^{\infty} \frac{z^n}{n!} = 1 + z + \frac{z^2}{2} + \ldots \tag{8}$$

To first order and under the scrutiny of mathematicians, the above expression can be rewritten in the limit

$$\lim_{v \to 0} \frac{1 - \exp\left(-2\pi i v\right)}{1 - \exp\left(-2\pi i (v/N)\right)} = \lim_{v \to 0} \frac{2\pi i v}{2\pi i (v/N)} = N. \tag{9}$$

The sum can be represented as N unit vectors in the complex plane forming a closed polygon for any integer value of v that is not a multiple of N and not 0. Alternatively one can also picture N points on the complex unit circle with azimuth $\phi = 2\pi(v/N)\tau$. The points are symmetrically distributed on the rim of the unit circle which is why their sum evaluates to 0. With v/N being a rational, τ will numerate all equally spaced points, separated by $\Delta\phi = 2\pi(v/N)$ and ends with a point at $\phi = 2\pi(v/N)(N-1)$ for $\tau = N - 1$ - which is again separated by $\Delta\phi = 2\pi(v/N)$ from the unit vector along the real axis. v is a measure for how many times the N points will wrap around - for $v = 1$, each point will be occupied only once, while $v = 2$ indicates that there are two points for each (cyclic) ϕ, etc...

(c) The power spectrum $S(v)$ for two sine waves with slightly different oscillation frequencies f is shown below. The x-axis is the frequency axis, in units of v/N, with v being an integer.

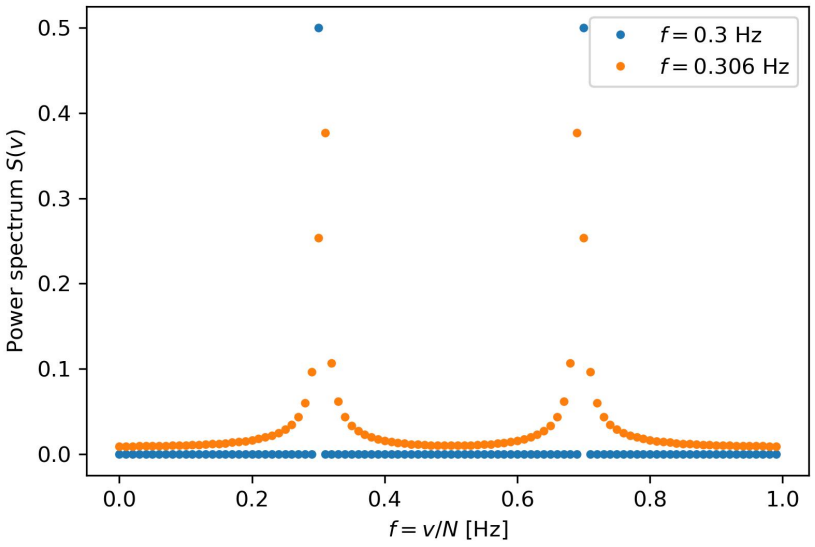

Figure 3: (Two-sided) power spectrum as a function of frequency in Hz.

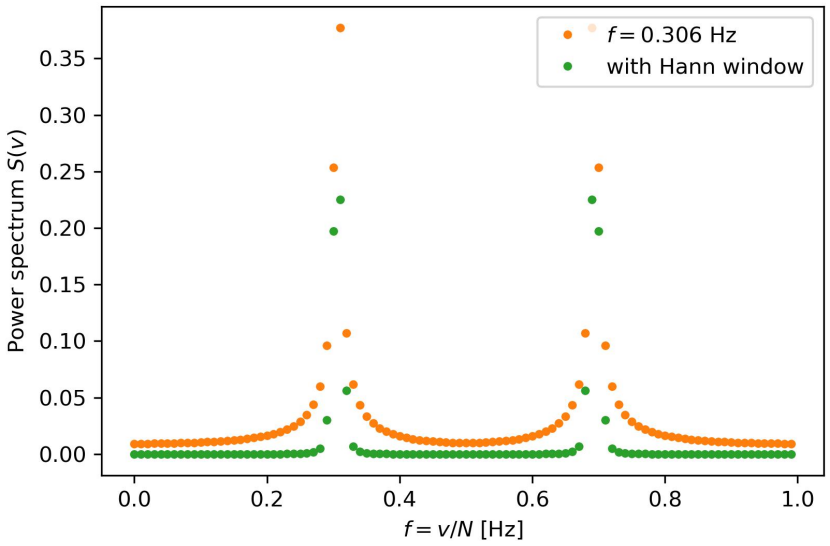

Figure 4: After multiplying the signal in real space with the Hann window function, the peaks are now narrower and spectral leakage is reduced at the cost of amplitude.

Hence, $v/N = 30/100 = 0.3$Hz is an integer frequency, while $v/N = 0.306$Hz is not. One can see two peaks due to the Nyquist theorem (which states that frequencies above $f_{\max}/2$ are mirrored from lower frequencies). In the non-integer case, the result is clearly not the "simple" discrete Dirac delta function any longer, but suffers from spectral leaking around the peak.

(d) To reduce spectral leakage the signal is multiplied with the Hann window function

$$w(\tau) = 0.5\left(1 - \cos\left(\frac{2\pi\tau}{N}\right)\right). \tag{10}$$

The result is shown in Fig. 4 - The speaktral leakage is indeed reduced dramatically, at the cost of an amplitude drop (albeit not as dramatic).

(e) The power spectrum of the Hann window is indeed (up to machine precission) the array `[N/2, N/4, 0, ..., 0, N/4]` (see Fig. 5).

5. (a) I wrote two helper functions to compute the power spectrum and smooth-en the data:

```
import numpy as np
from matplotlib import pyplot as plt
import h5py
from scipy import signal
import json
import os

def power_spectrum(arr, window):
N = len(arr)
# plt.plot(arr)
window = signal.get_window(window, N)
```

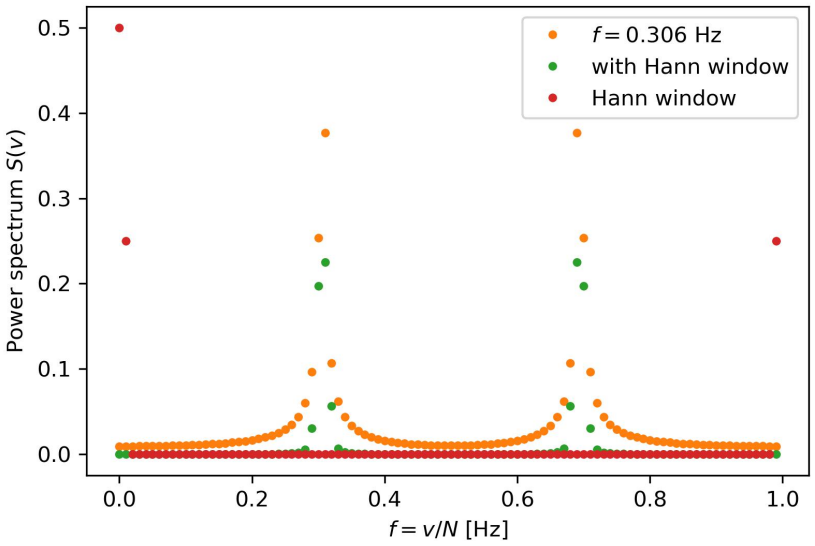

Figure 5: DFT of the Hann window

```
    windowed_arr = arr*window
    spectrum = np.fft.fft(windowed_arr)
    return np.abs(spectrum)**2

def smooth_data(arr, sigma):
    # this is basically jons code
    N = len(arr)
    tau = np.arange(N)
    tau[N//2:] = tau[N//2:] - N
    gauss = (1/sigma*np.sqrt(2*np.pi))**2*np.exp(-0.5*((tau)/sigma)**2)
    arr_ft = np.fft.fft(arr)
    gauss_ft = np.fft.fft(gauss)
    return np.fft.ifft(arr_ft*gauss_ft)
```

I then convoluted the raw data from the two LIGO detectors for each event with a windowing function and computed the power spectrum. I chose the `blackman` window. Subsequently, the power spectrum is further convoluted with a Gaussian to smoothen out the data. The noise models are shown in Fig. 6.

```
    # get all events stored in BBH_events dict
    events=json.load(open(source))
    window = 'blackman'
    sigma = 10

    for event in events:
        file_name = os.path.join(events[event])

        strain1,dt1,utc1 = read_file(file_name['fn_H1'])
        Hanford = power_spectrum(strain1, window)
```

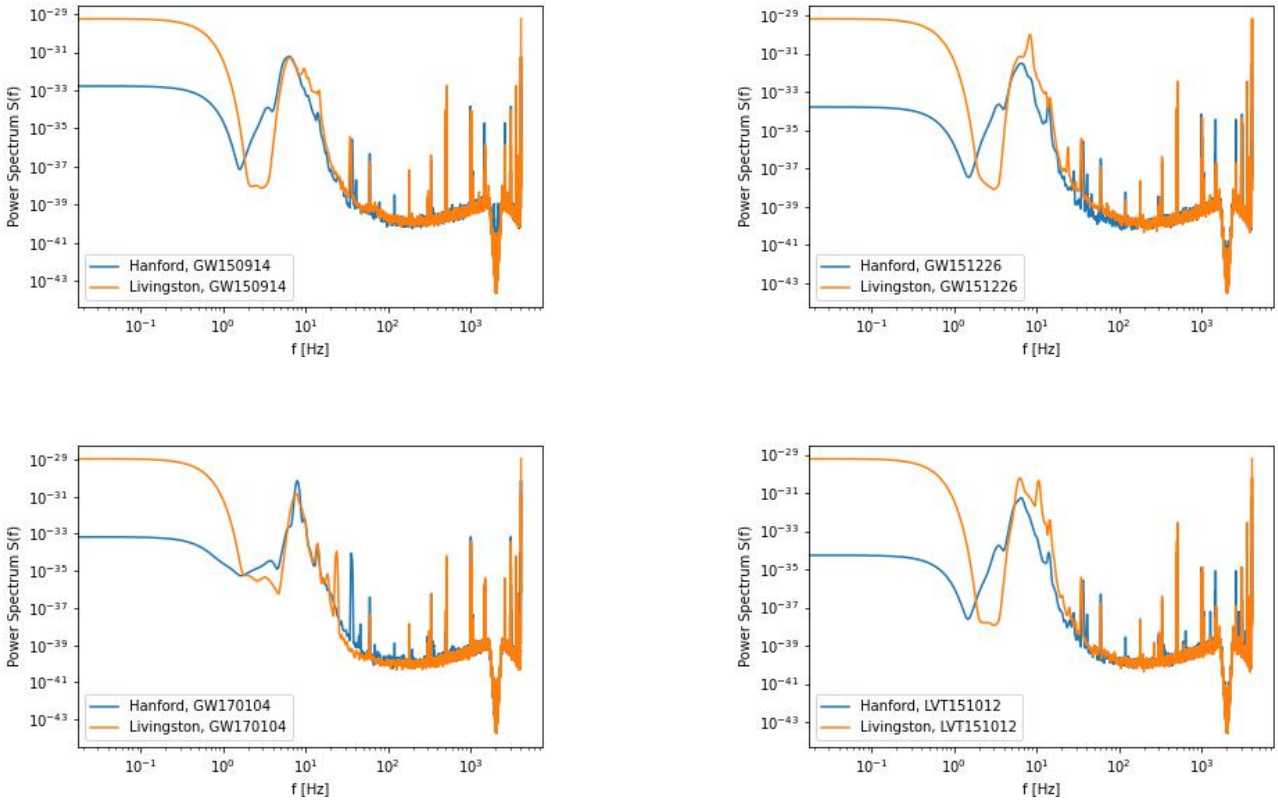

Figure 6: Smoothened power spectrum for all four events.

```
strain2,dt2,utc2 = read_file(file_name['fn_L1'])
Livingston = power_spectrum(strain2, window)

# calculate frequency in Hz
N_Han = len(Hanford)
N_Liv = len(Livingston)

f_Han = np.arange(N_Han)/(N_Han*dt1)
f_Liv = np.arange(N_Liv)/(N_Liv*dt2)

# smooth data
smooth_Han = smooth_data(Hanford, sigma)
smooth_Liv = smooth_data(Livingston, sigma)
```

(b) With the determined noise models, I applied a matched filter to each event. The results (Fig. 7) show "spikes" close to the beginning of their axes. The fact that the amplitude seems to thin out around the center is potentially an artifact and the whole array might need "recentering", but I was not entirely sure if that is correct. The code in the `for`-loop above needed to be supplemented with:

```
# PART (b)
```

```
# get templates for matched filtering
tp, tx = read_template(file_name['fn_template'])
N = len(tp)
window = signal.get_window(window, N)

# noise matrices
Ninv_Han = 1/smooth_Han
Ninv_Liv = 1/smooth_Liv

# whiten data
white_Han = np.sqrt(Ninv_Han)*np.fft.fft(strain1*window)
white_Liv = np.sqrt(Ninv_Liv)*np.fft.fft(strain2*window)

# get template ft and whiten
tp_ft = np.fft.fft(tp*window)
white_temp_Han = np.sqrt(Ninv_Han)*tp_ft
white_temp_Liv = np.sqrt(Ninv_Liv)*tp_ft

rhs1 = np.fft.ifft(np.conj(white_temp_Han)*white_Han)
rhs2 = np.fft.ifft(np.conj(white_temp_Liv)*white_Liv)
```

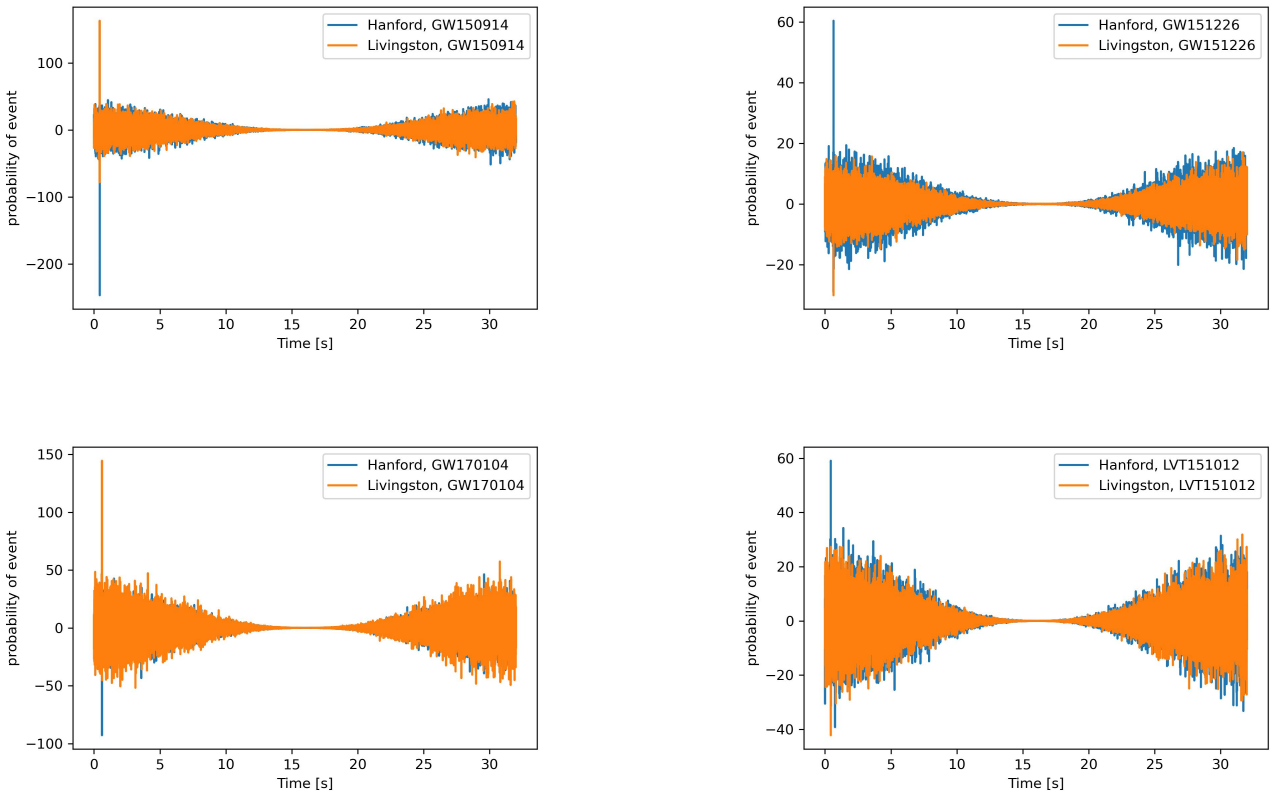

Figure 7: Matched filtering plots