Gravitational Wave Data Analysis

Cosmic Messengers Lab (PHY/AST-3880)

By: **Esha Sajjanhar**

Prof. Dipankar Bhattacharya
Submission Date: December 15, 2024

1 Data and Software Used

We use the Python packages `gwpy, gwosc` to obtain and process gravitational wave data from LIGO. The analysis here follows the tutorials from the 2023 Gravitational Waves Open Data Workshop.

2 Analysis

2.1 Obtaining Data

We begin by obtaining the LIGO data using `gwosc`. We first print a list of all available catalogs, then find all events within the GWTC-1 catalog catalog and all strain datasets within that.

```
import gwosc
from gwosc.datasets import find_datasets
from gwosc import datasets

#-- List all available catalogs
print(find_datasets(type="catalog"))

#-- Print all the GW events from the GWTC-1 catalog
gwtc1 = datasets.find_datasets(type='events', catalog='GWTC-1-confident')
print('GWTC-1 events:', gwtc1)

#-- Print all the large strain data sets from LIGO/Virgo/KAGRA observing runs
runs = find_datasets(type='run')
print('Large data sets:', runs)
```

We can also use keywords within these commands to further limit our search results. We can also find the gps time of events or find events using their gps time stamp.

```
from gwosc.datasets import event_gps
gps = event_gps('GW190425')
print(gps)
from gwosc.datasets import event_at_gps
print(datasets.event_at_gps(1240215503))
from gwosc.datasets import run_segment
print(run_segment('O1'))
```

```
(run_segment('O1')[1]-run_segment('O1')[0])/(3600*24)
from gwosc.datasets import run_at_gps
print(run_at_gps(1240215503))

O1_events = datasets.find_datasets(type='events', catalog='GWTC-1-confident',
    segment=run_segment('O1'))
print(O1_events)
```

We can also find the url of at which a dataset is available and filter by merger parameters.

```
from gwosc.locate import get_event_urls
urls = get_event_urls('GW150914')
print(urls)

urls = get_event_urls('GW150914', duration=32, detector='L1')
print(urls)

from gwosc.datasets import query_events
selection = query_events(select=["25 <= network-matched-filter-snr <= 30"])
print(selection)
```

Finally, we can import the timeseries of any dataset thus identified and plot the signal as shown below.

```
from gwpy.timeseries import TimeSeries
ldata = TimeSeries.fetch_open_data('L1', *segment, verbose=True)

plot = ldata.plot()
```

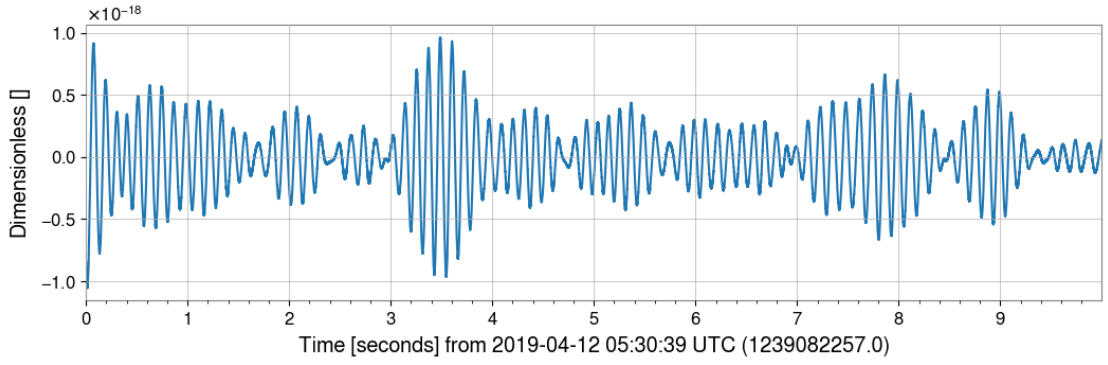

Figure 1: Time series data from the L1 detector for the event GW150914.

2.2 Fourier Transform of the Data

Once we have obtained the timeseries data, we want to perform a Fourier transform in order to further identify features of the data. However, the edges of the data can often affect the quality of the FFT which assumes periodicity. Thus, we apply a window to our data before performing the FFT. We then plot the amplitude of the frequency series that we obtain after performing the transform.

```
from scipy.signal import get_window
window = get_window('hann', ldata.size)
lwin = ldata * window

fftamp = lwin.fft().abs()
```

```
plot = fftamp.plot(xscale="log", yscale="log")
plot.show(warn=False)
```

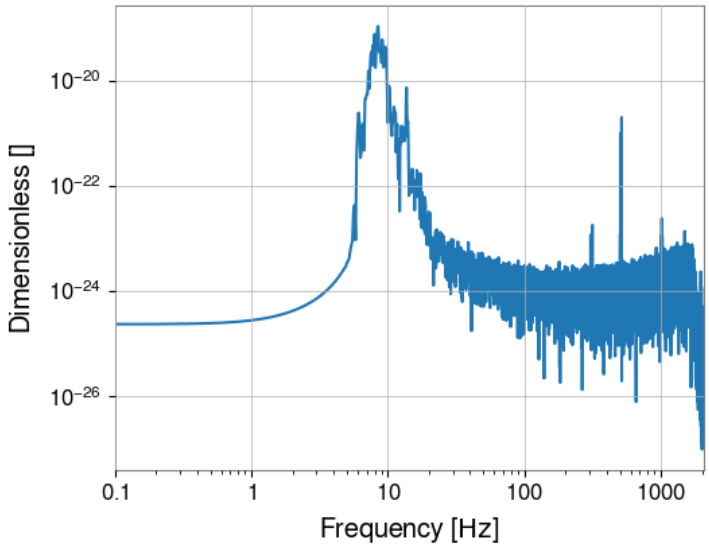

Figure 2: Amplitude of the frequency series obtained after performing windowing.

The fluctuations seen above 10 Hz are intrinsic noise and thus appear to be random. We can remove these by averaging the squared moduli of the Fourier components. This gives us the Power Spectral Density (PSD). If we express the result not as power but as an amplitude, we can obtain the Amplitude Spectral Density (ASD). This can be obtained

```
asd = ldata.asd(fftlength=1, method="median")
plot = asd.plot()
plot.show(warn=False)
```

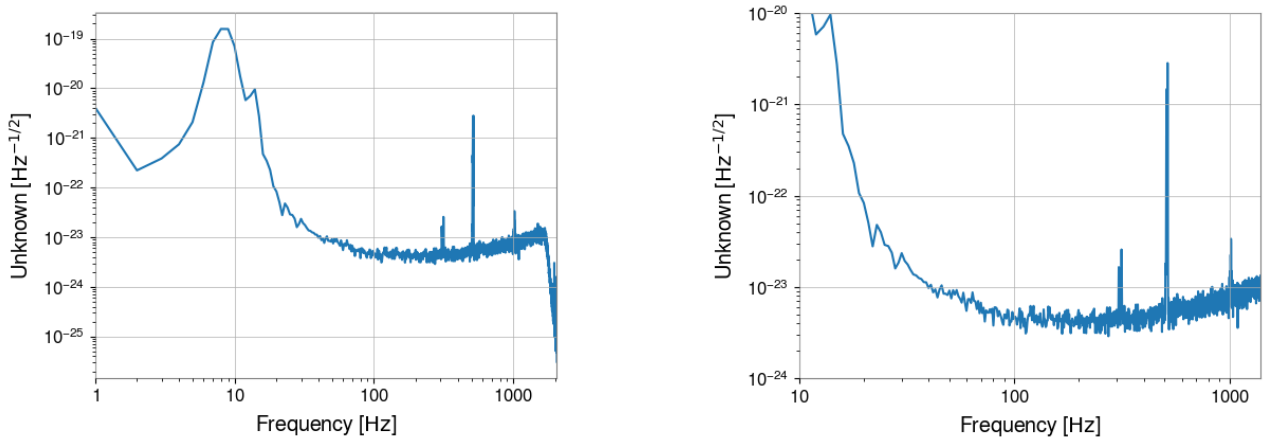

Figure 3: *Left:* The Amplitude Spectral Density (ASD) over the full frequency range. *Right:* ASD zoomed in after the seismic noise shoulder at 10 Hz and before the Nyquist frequency.

However, there is still a lot of noise in the ASD. We can now load more data and average it to obtain a cleaner ASD.

```
ldata2 = TimeSeries.fetch_open_data('L1', int(gps)-512, int(gps)+512, cache=True)
lasd2 = ldata2.asd(fftlength=4, method="median")
plot = lasd2.plot()
```

```
ax = plot.gca()
ax.set_xlim(10, 1400)
ax.set_ylim(1e-24, 1e-20)
plot.show(warn=False)
```

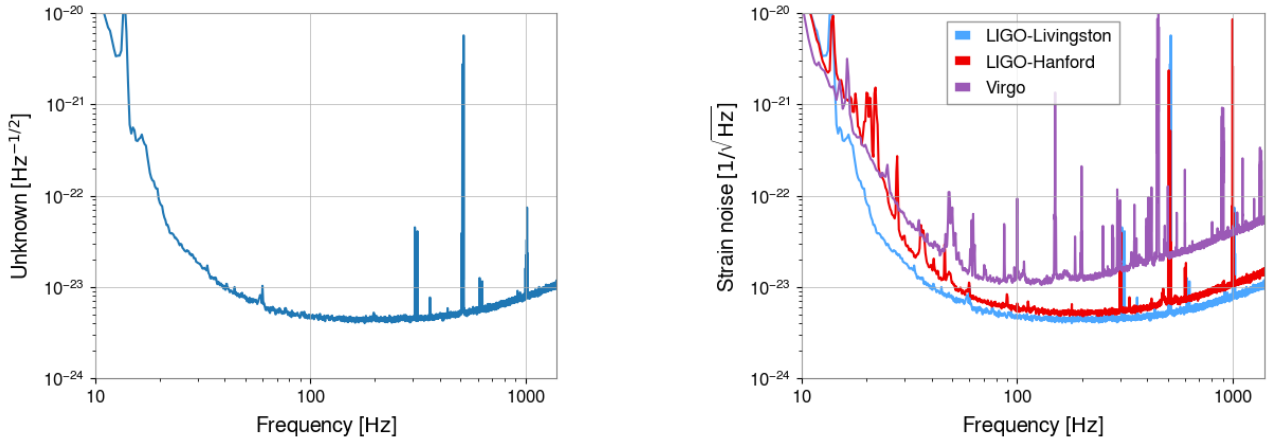

Figure 4: *Left:* The Amplitude Spectral Density (ASD) for Livingston, averaged over larger data segments. *Right:* Averaged ASD for the Livingston, Hanford and Virgo data.

2.3 Generating Waveforms

In order to identify an event, we first generate a sample waveform known as a 'template' and then search for signals which match that template. Here, we use 'IMRPhenomD' as a model approximant to the gravitational wave.

```
from pycbc.waveform import td_approximants, fd_approximants
hp, hc = get_td_waveform(approximant="IMRPhenomD",
                         mass1=10,
                         mass2=10,
                         delta_t=1.0/16384,
                         f_lower=30)
```

This gives us the waveform shown below. Here, the two polarizations differ only by their phase.

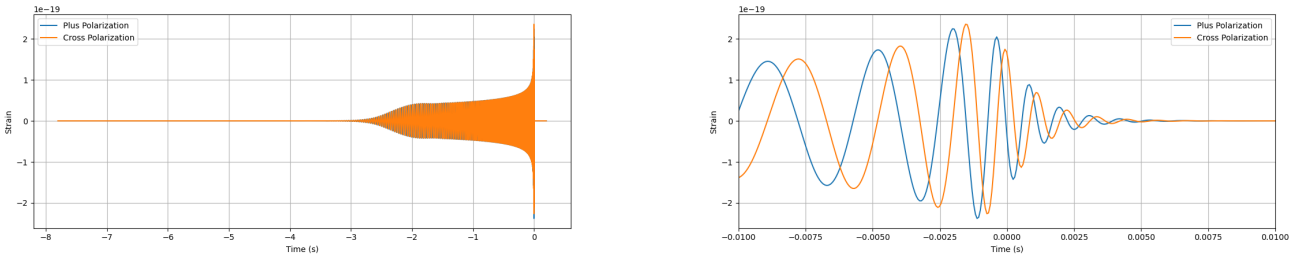

Figure 5: Template waveforms generated using the approximant 'IMRPhenomD'.

2.4 Matched Filtering

We now use the library `PyCBC` for matched filtering. This involves calculating a cross-correlation in both the time and the frequency domains. This also requires identifying and removing the noise in the signal.

Colored noise refers to noise which has a frequency-dependent variance while white noise refers to noise that is frequency independent. To detect a signal buried in colored noise, we can use `PyCBC`. Here, we have used it to estimate modelled colored noise as a demonstration.

```
import pycbc.noise
import pycbc.psd

flow = 10.0
delta_f = 1.0 / 128
flen = int(sample_rate / (2 * delta_f)) + 1
psd = pycbc.psd.aLIGOZeroDetHighPower(flen, delta_f, flow)

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

#estimating the colored noise using PyCBC
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)
```

As shown in the figure below, this gives us a good approximation to the colored noise in the signal.

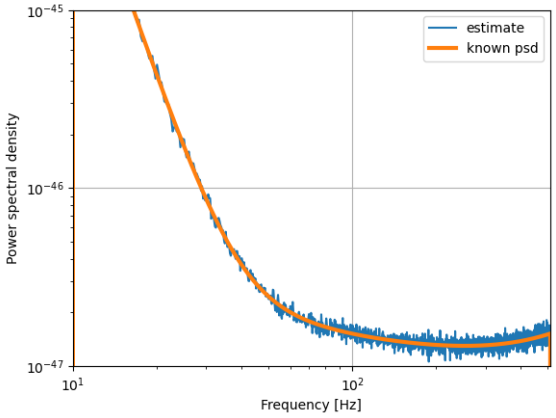

Figure 6: Approximation to the colored noise produced using PyCBC.

Evidently, this produces a good fit to the noise and we can use this fit to remove noise from event data. In order to do this, we use the noisy data from the detector and divide it by the PSD. We can then perform cross-correlation between the signal and the template waveform.

We now execute this for real data for the merger GW150914 using data from the Hanford observatory. The time series has been shown below. The low frequency noise has been removed from this data. However, the highpass filter causes a strange spike at the boundries. This is because the data is not cyclic. We can fix this by having the data wrap around at the boundaries. Next, we find the power spectral density and generate a template waveform. In this case, we use the approximant 'SEOBNRv4_opt'. We also shift the data so that the merger corresponds to the first bin of the data. This is done so that the template and the data correspond to the same parts of the data bin-wise. Finally, we calculate the cross-correlation between these. This cross-correlation has been plotted below.

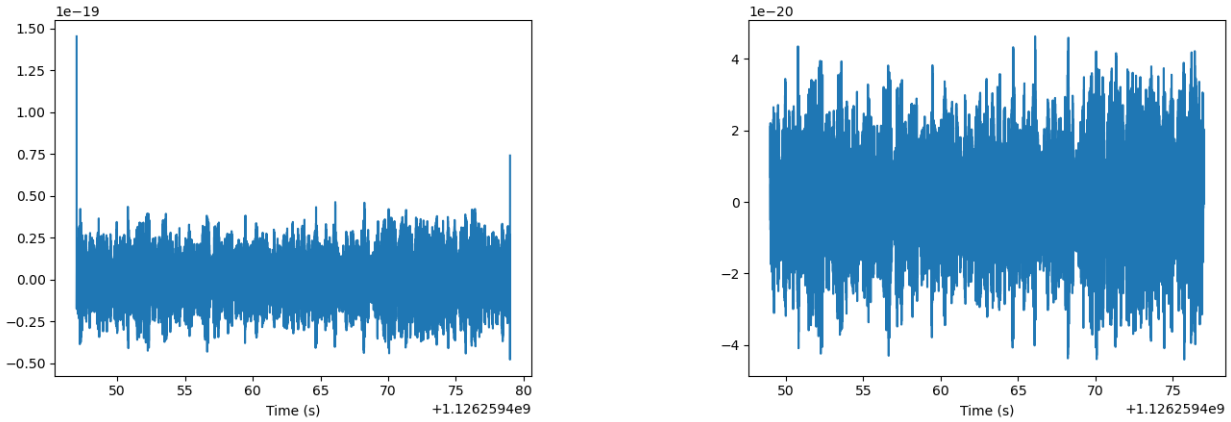

Figure 7: H1 time series for GW150914. *Left:* After downsampling and applying high pass filter. *Right:* After having the data wrap around the boundaries.

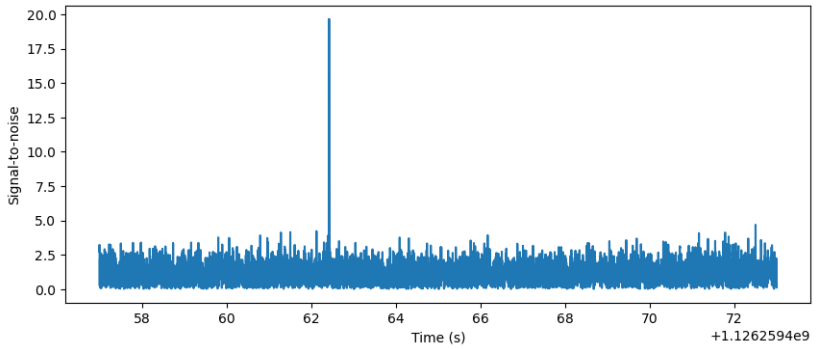

Figure 8: Cross-correlation between the template and the signal.

```
from pycbc.filter import matched_filter
import numpy

snr = matched_filter(template, conditioned,
                     psd=psd, low_frequency_cutoff=20)
snr = snr.crop(4 + 4, 4)
```

We found a signal at ∼1126259462.4 s with SNR ∼19.68. We can now compare the whitened data and the template.

```
white_data = (conditioned.to_frequencyseries() / psd**0.5).to_timeseries()
white_template = (aligned.to_frequencyseries() / psd**0.5).to_timeseries()

white_data = white_data.highpass_fir(30., 512).lowpass_fir(300, 512)
white_template = white_template.highpass_fir(30, 512).lowpass_fir(300, 512)

white_data = white_data.time_slice(merger.time-.2, merger.time+.1)
white_template = white_template.time_slice(merger.time-.2, merger.time+.1)
```

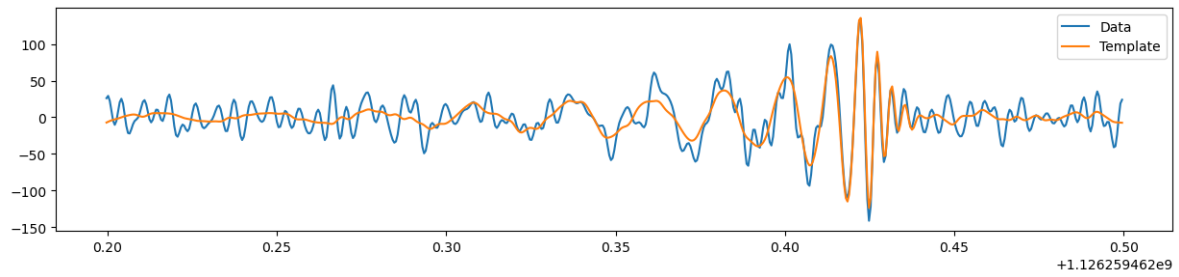

Figure 9: Comparision between template and whitened event data.

2.5 Q-transform

FInally, we can perform a Q-transform of our data. A Q-transform helps us visulaise the time variation of a PSD. It thus helps us understand the evolution of the merger signal. Q-transforms are time-frequency representations of the signal and thus use both the time series and the PSD.
We create Q-transforms for the data as well as for the template-subtracted data in order to see how well our template describes the observed data.

```
subtracted = conditioned - aligned

for data, title in [(conditioned, 'Original H1 Data'),
                    (subtracted, 'Signal Subtracted from H1 Data')]:

    t, f, p = data.whiten(4, 4).qtransform(.001, logfsteps=100, qrange=(8, 8),
    frange=(20, 512))
    pylab.figure(figsize=[15, 3])
    pylab.title(title)
    pylab.pcolormesh(t, f, p**0.5, vmin=1, vmax=6, shading='auto')
    pylab.yscale('log')
    pylab.xlabel('Time (s)')
    pylab.ylabel('Frequency (Hz)')
    pylab.xlim(merger.time - 2, merger.time + 1)
    pylab.show()
```

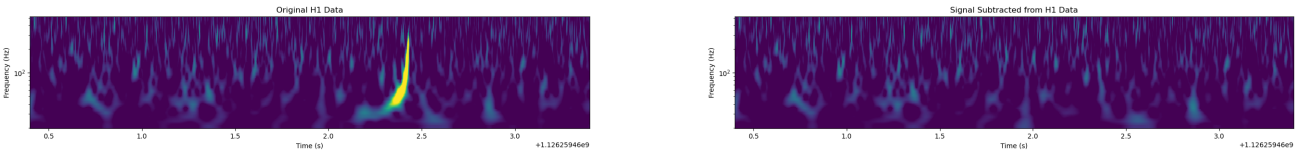

Figure 10: *Left:* Q-transform of the H1 data for the event GW150914. *Right:* Q-transform of the template-subtracted data for the same event.

We see a very distinct 'chirp' waveform in the Q-transform of the data. No clear signal can be seen in the template-subtracted data. This allows us to conclude that the template indeed describes the data well.

2.6 Signal Significane

Now, we want to characterise the SNR of our data. Here, we will be using data from the event GW170814. To begin with, we perform all the steps described above- we create a PSD, whiten

the data, generate a template waveform, and perform cross-correlation. We then use the libtrary `pycbc.vetoes` to find the signal-to-noise ratio (SNR) of our data.

```
from pycbc.vetoes import power_chisq

chisq = {}
for ifo in ifos:
    nbins = 26
    chisq[ifo] = power_chisq(hp, data[ifo], nbins, psd[ifo], low_frequency_cutoff
    =20.0)
    chisq[ifo] = chisq[ifo].crop(5, 4)

    dof = nbins * 2 - 2
    chisq[ifo] /= dof
```

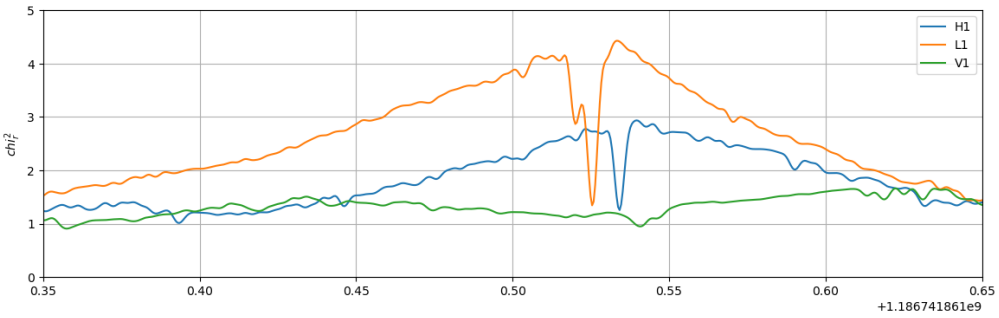

Figure 11: χ^2_r time series for GW170814 detection using 3 detectors.

The dip seen in the centre of the timeseries is because the template describes the signal well and aligns with the data in this region. We can re-weight the SNR so that it does not have this negative peak. Wherever $\chi^2 > 1$, we redefine the SNR ρ as,

$$\hat{\rho} = \frac{\rho}{\frac{1}{2}[1 + (\chi^2_r)^3]^{1/6}}. \tag{1}$$

```
from pycbc.events.ranking import newsnr

nsnr = {ifo:newsnr(abs(snr[ifo]), chisq[ifo]) for ifo in ifos}

for w, title in [(8, 'Wide View'), (.15, 'Close to GW170814')]:
    pylab.figure(figsize=[14, 4])
    for ifo in ifos:
        pylab.plot(snr[ifo].sample_times, nsnr[ifo], label=ifo)

    pylab.legend()
    pylab.title(title)
    pylab.grid()
    pylab.xlim(m.time - w, m.time + w)
    pylab.ylim(0, 15)
    pylab.xlabel('Time (s)')
    pylab.ylabel('Re-weighted Signal-to-noise')
    pylab.show()
```

We now find that the signal to noise ratio peaks at the time when the event was observed which can be easily distinguished from the transient noise peaks.

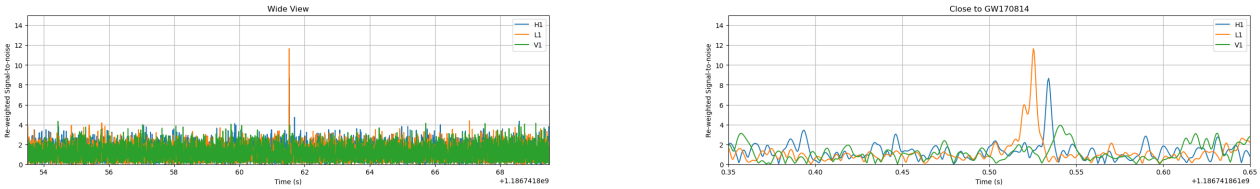

Figure 12: *Left:* Reweighted SNR over the entire range. *Right:* Reweighted SNR for the signal only.

3 Conclusion

The procedure for finding and characterising an event in gravitational wave data has been described above. In general, after a template waveform has been selected, we can find the properties of the merger using Bayesian parameter estimation. This allows us to ascertain the distance to and the masses of the components of the binary merger which could have produced the observed waveform.

4 References

Gravitational Wave open data workshop 6 https://github.com/gw-odw/odw-2023.