An Analysis of Gravitational Wave Events

Daniel Bautista - 3035130645

Astro C161 - Final Project

May 9, 2020

Background

In 1916, Albert Einstein published his General Theory of Relativity, which describes how spacetime curves in the presence of mass and energy. With it, came predictions of the existence of black holes and gravitational waves. Gravitational waves are disturbances in spacetime that come with the acceleration of massive objects. In 2016, the Laser Interferometer Gravitational-Wave Observatory (LIGO) announced the first direct observation of gravitational waves, with many more events being detected thereafter. In this project, I will analyze LIGO data for the GW150914 and GW170814 events which occurred during LIGO's first and second observing runs (The LIGO Scientific Collaboration and the Virgo Collaboration et al., 2018). I will use the Newtonian approximation for the interaction of two compact objects, which works well when the black holes are far apart, but will break down as they come closer. With this in mind, I will look at how well the Newtonian approximation holds up, compared to the full General Relativity results from LIGO.

The idea for this project was inspired by Problem Set #6 in Astro C161. In the Newtonian approximation, as two black holes spiral in towards each other before eventually merging, they will generate gravitational waves at a frequency given by:

$$f(t) = \frac{f_0}{(1 - t/t_m)^{3/8}} \tag{1}$$

where $f_0 = \sqrt{GM}/\pi a_0^{3/2}$ is the initial frequency of the orbit with initial separation a_0, and $t_m = \frac{5}{256}\frac{a_0^4 c^5}{G^3 M m_1 m_2}$, giving the time of the merger in terms of the total mass $M = m_1 + m_2$, the masses of the individual black holes m_1, m_2, and their initial separation. This equation predicts that the frequency of the waves will increase as t approaches t_m. This sharp increase in the frequency is seen in LIGO data, and is known as the "chirp." The chirp can allow us to constrain the masses of the black holes in a quantity known as the chirp mass. The chirp mass is described by:

$$M_{chirp} = \frac{c^3}{G}\left[\frac{5}{96}\pi^{-8/3} f^{-11/3} \frac{df}{dt}\right] \tag{2}$$

and is defined as:

$$M_{chirp} = \frac{(m_1 m_2)^{3/5}}{(m_1 + m_2)^{1/5}} \tag{3}$$

where f is a frequency in the chirp and $\frac{df}{dt}$ is the rate of change of the frequency with respect to time, at the time of the given frequency.

Methods

Cleaning the Data

The raw LIGO data is dominated by noise from the detector and its surroundings, as seen in the plots below:

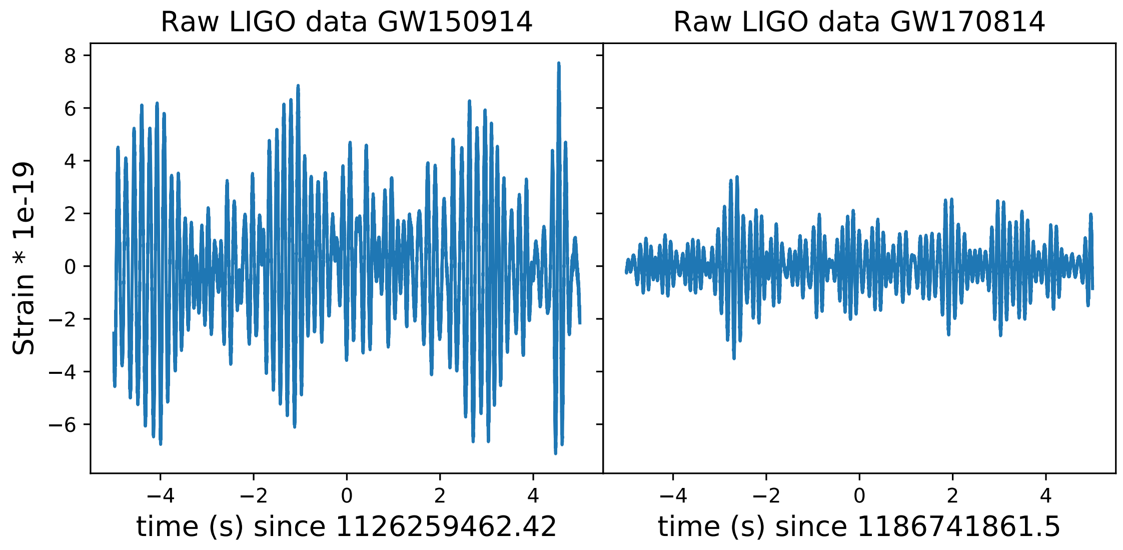

Figure 1

In order to remove noise from the data, it is necessary to know which frequencies are in the data and how much each frequency contributes. To do this, we will take the fourier transform of the data and plot the magnitude against the frequency. Below are the amplitude spectral density (ASD) for each event. In order to calculate the ASD, one must first find the power spectral density (PSD), which is the average of the square of the discrete fourier transform of the data, and from there, take the square root. The PSD was calculated using Python's `matplotlib.mlab.psd` function.

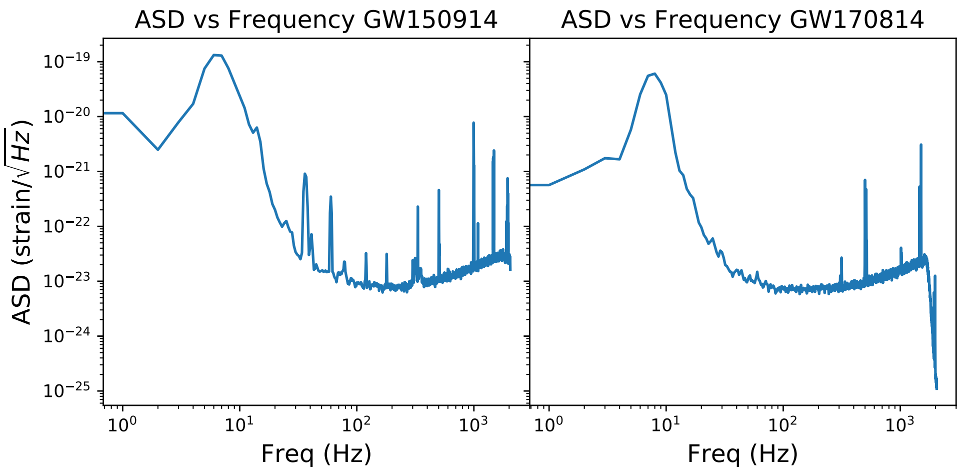

Figure 2

Since this plot is an average of 32 seconds of data, we can assume that it is dominated by instrumental noise, as the gravitational wave signal lasts for a fraction of a second. From here, we use a technique known as "whitening." Whitening the data is done the fourier domain, in which the fourier transform of the data is divided by the ASD and then transformed back. This is effective in suppressing the high power, lower frequency noise that comes from the instrument, but there is still high frequency noise obscuring the gravitational wave signal. To remove this, a technique called bandpassing is used. Bandpassing is a method of filtering the data based on frequency, and only allowing through frequencies within a certain range and rejecting those outside. In my banspassing function, I used Python's `scipy.signal.butter`, a Butterworth digital/analog filter design commonly referred to as a "maximally flat magnitude filter" and `scipy.signal.filtfilt`, which applies the filter twice, once forward and once backwards, to give a filter that has zero phase and a filter order twice that of the original.

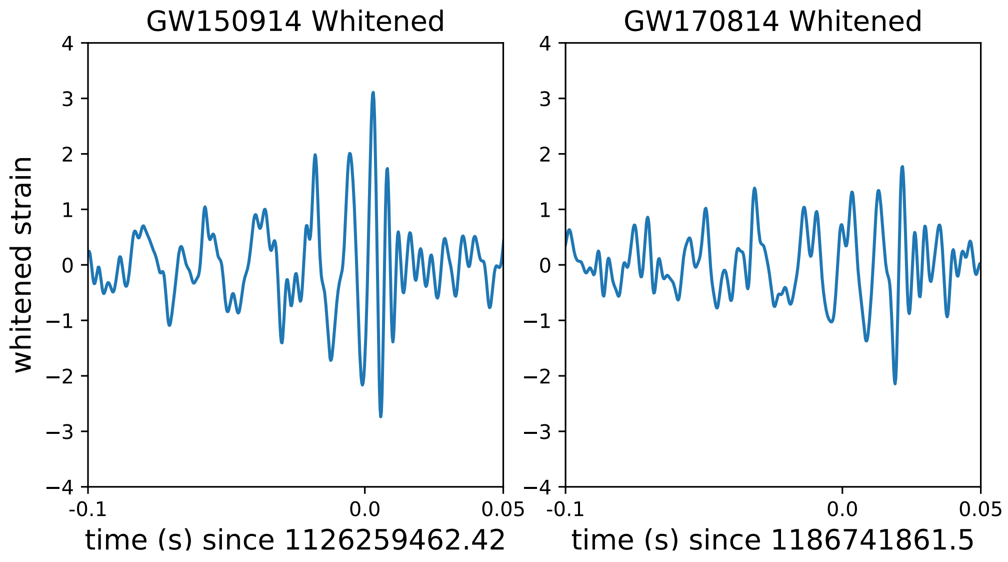

Figure 3

Above is the whitened and bandpassed data. Notable features are an increase in signal frequency and amplitude as the time approaches $t = 0$ (time of merger) and a drop in signal amplitude shortly after $t = 0$.

The Chirp

Now that the data has been cleaned so that the gravitational wave signal is apparent we can begin analyzing the chirp. The first step is to make a waterfall plot, showing the frequency vs time, with the intensity being represented by the color.

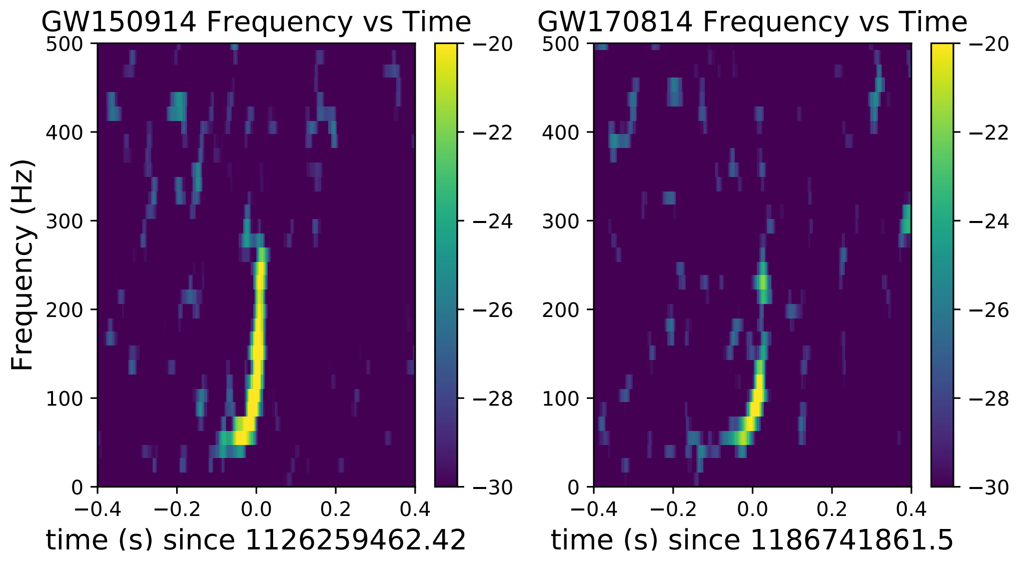

Figure 4

The chirp is clearly visible, centering around $t = 0$, as seen in the strain data in Figure 3, starting with a frequency around 50Hz and peaking just below 300Hz. The waterfall plot was generated using Python's `matplotlib.pyplot.specgram` function and was normalized to only show frequencies with a weight between -30 and -20. As described in equation (2), the chirp mass is dependent on the frequency and the rate of change of the frequency at that time.

In order to find a frequency and slope in the chirp, it will be necessary to fit a curve to the chirp. The first step will be to make our own frequency vs time scatter plot. This will be done by picking a small interval of time to perform a fourier transform to and "sliding" that window along the length of the signal. A small window is ideal because if the window is too large, the rapid change in frequency will be missed. After some trial and error, I found that the best window size ranged between 0.018 seconds and 0.244 seconds, which corresponded to segments of data ranging from 75 data points up to 1000 data points. After passing the data through this function, I was able to pick out the frequency with the greatest weight in fourier space, which corresponds to the dominant frequency in that window. Taking the midpoint of the time window gives me the time coordinate for the associated dominant frequency.

Analysis

After finding the dominant frequency for an interval of time, I need to select only the interval in which the chirp happens, otherwise there will be too much noise to fit a curve to my points. My boundaries for the window were frequencies between 1Hz and 300Hz and times between $t_m - 0.1$ seconds and $t_m + 0.05$ seconds.

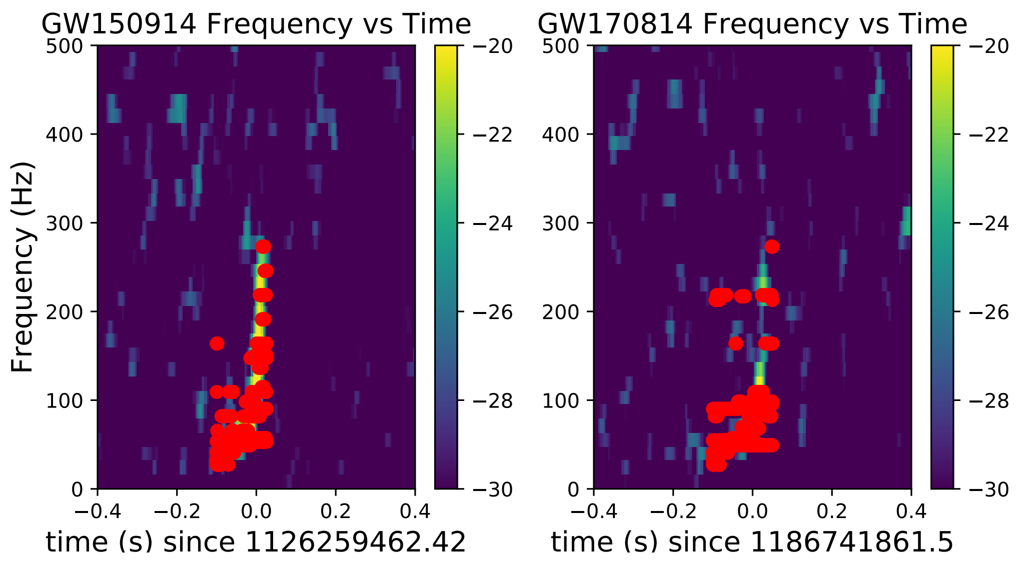

Figure 5

Above is the scatter plot of the times and dominant frequencies on top of the waterfall plot from Figure 4. The scatter plot is tightly grouped for the lower frequencies, with less points present as the frequency increases. A curve that is plotted to fit these points will most accurately describe the low frequency behavior, with the accuracy decreasing as the frequency rises. Due to the nonlinear behavior of the chirp, I will try a linear fit as well as a quadratic fit, and see which better fits the chirp. The fitting was done using Python's `numpy.polyfit` functions, which applies a first and second order polynomial fit to the data.

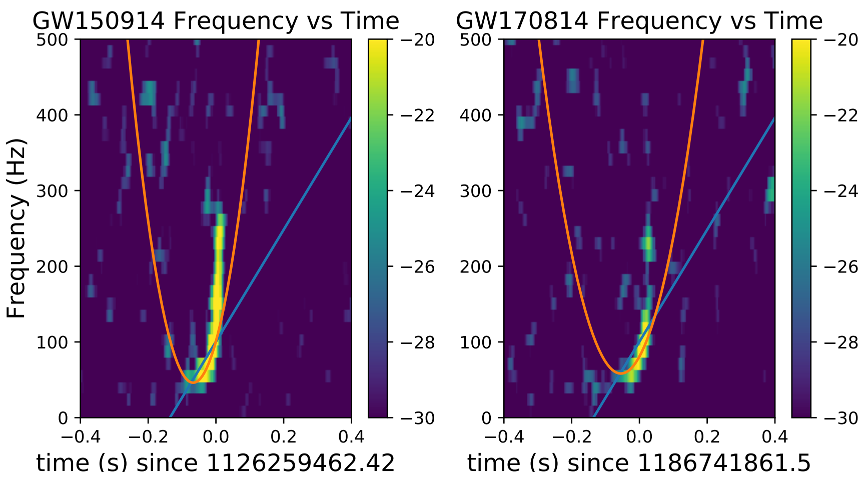

Figure 6

The quadratic fit does a much better job describing the behavior of the frequency, when compared to the linear fit. And as expected, the parabola tightly fits the low frequency behavior, but its accuracy diminishes as the frequency grows. The linear fit does a worse job describing the behavior of the frequency for all except a small interval, in which it briefly describes the average change between two frequencies. A benefit is that the linear fit intersects the parabolic fit in the low frequency range ie., where the parabola best describes the frequency.

As a result, these two plots can be used to find the frequency, f, and change in frequency, $\frac{df}{dt}$, treating them like a system of two equations with two unknowns. Due to the intersection of the two fits, we know that the slope of the linear fit must describe the slope of the parabola somewhere between the points of intersection. The most general forms of the curves are given in the table below:

Order	Equation	Derivative
1	$y = mx + b$	$y' = m$
2	$y = \alpha x^2 + \beta x + \gamma$	$y' = 2\alpha x + \beta$

Setting the two derivatives equal gives:

$$x = \frac{m - \beta}{2\alpha} \tag{4}$$

where x gives the point where the slope of the parabola is equal to the slope of the line. Plugging this point into the equation for the parabola gives the value of the parabola for the given slope. When we apply this to the problem at hand, of describing the behavior of the frequency, we are now able to determine the frequency f and its rate of change, $\frac{df}{dt}$. Now we can plug these two values into equation (2) which sets the chirp mass.

The results are:

Event	LIGO Result	Derived Result	Percent Error
GW150914	$28.6M_\odot$	$39.3M_\odot$	37.4%
GW170814	$24.2M_\odot$	$22.2M_\odot$	8.2%

These results are quite good, considering that they were derived using a Newtonian approximation. This is due to the fact that the frequency measurements were made relatively early during the merger, where the Newtonian approximation holds (LIGO Scientific and VIRGO Collaborations et al., 2017). In order reach a result even closer to the value calculated by LIGO, perturbation theory is needed to find the leading order correction factors from General Relativity (Kasen, 2020). Possible sources of error lie in the fitting of times and frequencies to the waterfall plot of the chirp. As seen in Figure 5, a majority of the points plotted fell on the chirp, however, there were several points that did not. These outliers were included in the data used to make the linear and quadratic regressions and shifted them away from having a better fit. As these regressions were used to derive the frequency and rate of change of the frequency, these errors carried over into the derived chirp mass.

Code

This program was executed in a conda environment set up by following the directions here. Below, I will paste and define the Python functions I used for this project.

```
def window_index(tevent, deltat, time):
    indxt = np.where((time >= tevent-deltat) & (time < tevent+deltat))
    return indxt
```

The `window_index` function locates the indices that satisfy the time boundaries in the list of time points in LIGO data and returns them as a list which can be used to pick out segments of data.

```
def asd(strain, NFFT, fmin, fmax):
    Pxx, freqs = mlab.psd(strain, Fs=fs, NFFT=NFFT)
    psd = interp1d(freqs, Pxx)
    return psd, Pxx, freqs
```

The `asd` function calculates the amplitude spectral density (ASD) by taking the square root of the power spectral density (PSD). It is used to break down the data into its frequency content. (Gravitational Wave Open Science Center, 2017)

```
def whiten(strain, interp_psd, dt):
    Nt = len(strain)
    freqs = np.fft.rfftfreq(Nt, dt)
    hf = np.fft.rfft(strain)
    white_hf = hf / (np.sqrt(interp_psd(freqs) /dt/2.0))
    white_ht = np.fft.irfft(white_hf, n=Nt)
    return white_ht
```

The `whiten` function fourier transforms the data, divides it by the ASD, which is also in the fourier domain, and then transforms back the result. This is done to suppress the low frequency noise and the large spikes from instrumental noise. (Gravitational Wave Open Science Center, 2017)

```
def bandpass(strain, fs):
    bb, ab = butter(4, [20.0*2/fs, 300.0*2/fs], btype="band")
    strain_bp = filtfilt(bb, ab, strain)
    return strain_bp
```

The `bandpass` function bandpasses the data, which is to say it passes the data through a Butterworth filter twice. Its purpose is to pass through only frequencies within a specified range and eliminate frequencies outside. (Gravitational Wave Open Science Center, 2017)

```
def max_value(arr):
    max_val = np.amax(arr)
    max_indx= np.where(arr == max_val)
    return max_indx
```

The max_value function takes in an array and returns the index of the entry with the greatest numerical value.

```
def spike_frequency(freqs, fourier_transform, mag=False):
if mag:
    pos = max_value(fourier_transform)
    return freqs[pos[0][0]]
pos = max_value(np.abs(fourier_transform))
return freqs[pos[0][0]]
```

The spike_frequency function takes in the raw fourier transform by default and picks out the dominant frequency using the max_value function.

```
def fourier_transform_freq_time(data, start_indx, interval, fs, dt):
    start = start_indx
    stop = start_indx+interval
    data = sfft.fft(data[start:stop])
    freqs= sfft.fftfreq(interval)*fs
    spike_freq = spike_frequency(freqs, data)
    time = (stop*dt + start*dt)/2
    return time, spike_freq
```

The fourier_transform_freq_time function takes in an array of data, a start index, and an interval on which to perform a fourier transform and returns the dominant frequency as well as the midpoint time for the interval using the spike_frequency function.

```
def transform_loop(data, interval, time_arr, freq_arr, fs, dt):
    for start_index in tqd(range(len(data)-interval),
                          leave=False, desc=str(interval)):
        time, freq = fourier_transform_freq_time(data, start_index,
                                interval, fs, dt)
        time_arr.append(time)
        freq_arr.append(freq)
        start_index = start_index + interval/16
    return time_arr, freq_arr
```

The transform_loop function takes in a large array of data and an interval on which to perform many small interval fourier transforms using fourier_transform_freq_time, and returns lists containing the time and dominant frequency for each interval.

```
def trim(time_arr, freq_arr, deltat, tmin, tmax, fmax):
    interval = np.where((time_arr >= deltat+tmin) & (time_arr < deltat+tmax))
    time_arr = time_arr[interval]
    freq_arr = freq_arr[interval]
    to_remove = []
    for i in range(len(time_arr)):
        if freq_arr[i] <= 0 or freq_arr[i] > fmax:
            to_remove.append(i)
    time_arr = np.delete(time_arr, to_remove)
    freq_arr = np.delete(freq_arr, to_remove)
    return time_arr, freq_arr
```

The `trim` function takes in the time and frequency lists and selects only the segment that is within the time boundaries. It then goes through the frequency list and records the entries that are outside the specified frequency boundaries. It then removes the frequency entries and their corresponding time entries and returns the shortened lists.

```
def derive_frequency(a, b, c, m):
    x = (m-b)/(2*a)
    f = a*x**2 + b*x + c
    return f
```

The `derive_frequency` function takes in the parameters for the linear and quadratic fits, and uses them to calculate a frequency where the slope of the parabola matches the slope of the line. This function is based on equation (4).

```
def chirp_mass(frq, dfdt):
    c = 3.0e8
    G = 6.67e-11
    m_sun = 1.989e30
    chirp_mass = (c**3.0/G) * ((5.0/96.0) * np.pi**(-8.0/3.0) *
                               frq**(-11.0/3.0) * dfdt)**(3.0/5.0)
    return chirp_mass/m_sun
```

Thh `chirp_mass` function is the Python equivalent of equation (2). It takes in the frequency and rate of change of the frequency and uses them to calculate the chirp mass.

The full Jupyter Notebook that I ran the code in was also attached on bcourses and is also available at `https://github.com/dbautista98/C161_Final_Project`, and contains the .hf5 files with the data.

References

Gravitational Wave Open Science Center
2017. Signal processing with gw150914 open data. `https://www.gw-openscience.org/GW150914data/GW150914_tutorial.html`.

Kasen, D.
2020. Astro c161 problem set 6. Inspiration for project.

LIGO Scientific and VIRGO Collaborations, B. Abbott, R. Abbott, T. Abbott, M. Abernathy, F. Acernese, K. Ackley, C. Adams, T. Adams, P. Addesso, et al.
2017. The basic physics of the binary black hole merger gw150914. *Annalen der Physik*, 529(1-2):1600209.

The LIGO Scientific Collaboration and the Virgo Collaboration, B. P. Abbott, R. Abbott, T. D. Abbott, S. Abraham, F. Acernese, K. Ackley, C. Adams, R. X. Adhikari, et al.
2018. Gwtc-1: A gravitational-wave transient catalog of compact binary mergers observed by ligo and virgo during the first and second observing runs. *Physical Review X*, 9.