School of Mathematics and Physics

Title:

Distinguishing Intermediate Mass Black Hole Mergers From Short Duration Glitches

Course: Physics, Astronomy, and Cosmology

Student No: UP881788

Year: 2021

School of Mathematics and Physics

Disclaimer

The enclosed project is entirely the work of the undergraduate student and any opinions enclosed are not necessarily those of any member of the staff of the School of Mathematics and Physics, University of Portsmouth. The text does not show any corrections of fact or calculations, and significant errors and omissions are possible. Any external reader or user of this report does so entirely at their own risk and responsibility, and neither the School of Mathematics and Physics, nor the University of Portsmouth can be held responsible for anything contained within this student project report.

Scientific Report

Distinguishing Intermediate Mass Black Hole Mergers From Short Duration Glitches

Jack Lloyd-Walters FRAS

UP881788

BSc (Honours) Degree in Physics, Astronomy, and Cosmology

School of Mathematics and Physics

2021

wordcount: 4830 words

Abstract

Glitches are a frequent occurrence with LIGO data, on the order of 10 an hour, and represent unwanted noise when searching for gravitational wave signals. Due to their similarity to IMBH merger events, they represent an obstacle to any search that deals with higher mass black holes. With a viable model of these glitch events, a full search could more easily distinguish IMBH mergers from short duration glitches. The glitch model created for this report was found to be accurate when searching against LIGO data using traditional matched filtering, and showed high similarity to events identified using the omicron scan, despite the difference in methods for detection. This glitch model was shown to not hinder a search for IMBH's, as merger templates for GW190521 still responded more strongly than the glitch model, showing it's safety in respect to true mergers. As such, this could pose a viable model for an extended search across a much larger swathe of LIGO data, with higher mass resolutions providing a logical improvement.

Acknowledgements

I would like to thank my supervisor, Dr Andrew Lundgren, for providing immeasurable support in furthering my understanding of the topic and the programming challenges I encountered throughout, as well as for providing feedback every week on whatever work I happened to show.

I would also like to thank the members of the GWastro Slack group for their assistance with the many PyCBC and Jupyter Issues I encountered while programming for this project, of which there were many, as well as providing a great audience for the presentation based on this report.

To that end, I of course extend my thanks to the entire PyCBC team, both for the tutorial documentation that allowed me to learn this package, and for the vast suite of functions that catered to almost any required programming task. Alongside this, my gratitude to Python, Numpy and Scipy, for their huge array of tools, and an enjoyable decade of programming.

Finally, I would like to thank the Ligo Collaboration for providing the server environment that allowed this search to run, as well as all data collected that have made Gravitational wave astronomy possible.

Contents

Disclaimer . . . . **II**

Abstract . . . . **IV**

Acknowledgements . . . . **IV**

1 Introduction **1**
- 1.1 The Intermediate Mass Black Hole Problem 1
- 1.2 Glitch Events 2
- 1.3 Moving forward 4

2 Theory **5**
- 2.1 Constructing a Glitch 5
- 2.2 Matched filtering 10

3 Methodology **12**
- 3.1 The Search 12
  - 3.1.1 Data collection 12
  - 3.1.2 Template Generation 13
  - 3.1.3 Signal Processing 14
- 3.2 Compiling results 15
  - 3.2.1 Event Detection 15
  - 3.2.2 Graphical Output 16

4 Results **16**

5 Conclusion **18**

6 Appendix **20**
- 6.1 Bibliography 20
- 6.2 List of Figures 21
- 6.3 Results 23
- 6.4 Search Program 27
  - 6.4.1 External Functions 27
  - 6.4.2 Search Script 41

1 Introduction

Note: A Github Repository containing the search code and other relevant data can be found here [9].

1.1 The Intermediate Mass Black Hole Problem

Since their inception a century ago, we have found a multitude of black holes spanning the very small, to the monstrously large. At the lowest range are stellar black holes, remnants of the largest stars whose mass is less than $10^2 M_\odot$. In contrast are the supermassive black holes, whose mass is sufficient to dominate the evolution of galaxies, and are thought to reside with their cores. Within this continuous range of known compact objects is an odd discontinuity; black holes whose mass ranges from $10^2 - 10^5 M_\odot$.

While a handful of candidates for these intermediate mass black holes (IMBH's) have been found, only a single one has ever been confirmed. This object, GW190521 [1], was found on the 19th of may 2019 following a detection trigger in multiple detectors, which was thought to result from the merger of two large stellar mass black holes. This merger event, characterised by a short duration and low peak frequency, sits squarely within the area that LIGO is most sensitive to, raising questions as to their observed scarcity.

As the largest stars reach the end of their lives, temperatures and pressures within their cores are sufficient for pair creation to play a dominant role in stellar evolution. As stars support themselves against gravitational collapse by way of radiation pressure, a portion of these photons becoming particle-antiparticle pairs destabilises the previously established equilibrium [6].

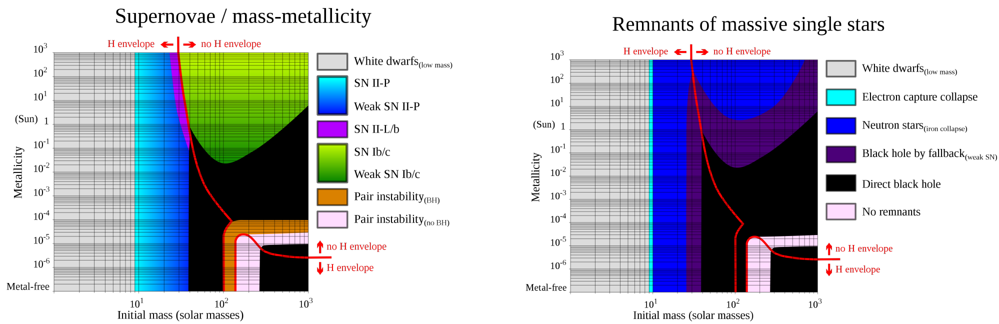

Figure 1: Supernovae types, and remnant object, given initial star mass and metallicity. credit: [3] [4]

For stars between 100 and $130 M_{\odot}$ this results in several pulsations, where increased pair production causes the star to contract, raising core fusion rate until a new equilibrium is established, with several solar mass of material ejected from the outermost layers of the star in the process. This continues until the star falls below the required limit for pair production, and evolves further as a regular (albeit massive) star.

Stars within the 130 and $250 M_{\odot}$ boundary experience a much more energetic suite of pair production events due to their increased pressures. While smaller stars can eventually reach a new equilibrium after the initial pair production, these stars experience a runaway feedback loop. Overpressure in the star is sufficient to completely consume the core as a seconds-long thermonuclear explosion, blowing apart the star in a highly destructive and energetic pair-instability supernova [6]. We can see this in figure 1 as a blank area in the two graphs.

Further massive stars, those above $250 M_{\odot}$ undergo photodisintegration before pair-production can completely consume the star. Photodisintegration is an endothermic (energy absorbing) process whereby a nucleus absorbs a gamma ray, enters an excited state, and immediately deexcites by emitting one or more subatomic particles. This prevents thermonuclear runaway, as distinct fusion processes require specific atomic isotopes, and the star eventually collapses completely in on itself to form a massive black hole [5]. While this is the expected evolutionary path of a star this massive, very few, if any, of these stars besides the very first in the universe are expected to have formed.

From this, we expect that intermediate mass black holes form only through gravitational mergers, though a secondary problem arises: Supermassive black holes. If our model of bottom up formation is correct, SMBH's form through mergers of massive seed black holes, either typical IMBH's or direct collapse black holes [8]. Given the high population of both SMBH's and stellar mass black holes, we should expect to see many remnants within the IMBH range.

1.2 Glitch Events

Within any arbitrary segment of gravitational strain data are glitch events. Glitches are, broadly speaking, short duration non-Gaussian wave-forms with similar spectral properties to actual merger events, though without an astro-

physical source, and an almost unlimited loudness. These occur frequently, on the order of ten an hour, and are independent between detectors, with a rare chance that two detectors may see a chance overlap of independent glitches.

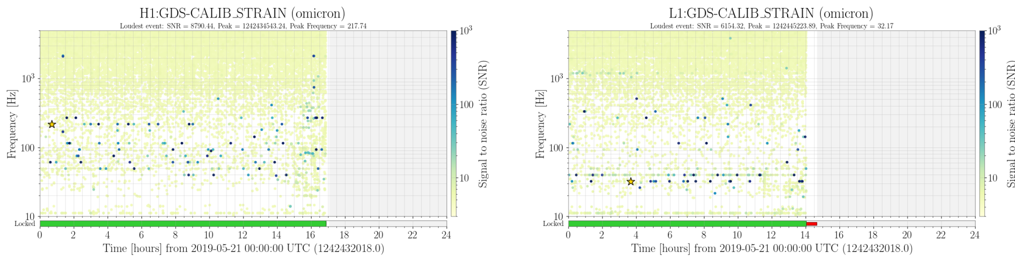

Figure 2: Omicron scan of the Hanford and Livingston detectors, demonstrating the frequency of glitch events

Over years of LIGO observation, we have seen an entire zoo [7] of glitches. To trim what would otherwise be a broad topic, the particular glitches that share features with the blip (**B**and **L**imited **I**m**P**ulse) glitches will be the main focus of this paper, with an example shown in figure 3. This figure shows a specific class of time-frequency diagram called the "QTransform" which shows the energy content of each frequency in the detector strain changes over time.

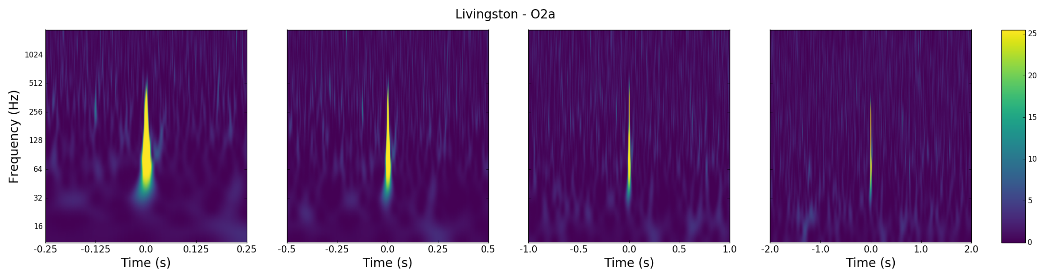

Figure 3: Sample Glitch event in the Livingston Detector. The colour scale is the normalised energy for this time range

From this plot we can see how short a duration blip glitches are compared to mergers, significantly less than a tenth of a second for this specific glitch. As it will be important later, we can also see that this glitch occurs between the 16 and 1024Hz range, with a greater proportion of the glitch occurring at the lower end of this frequency range.

To further visually distinguish glitches from mergers, figure 4 shows the spectral plot of a merger event. Note how this event is asymmetric, unlike

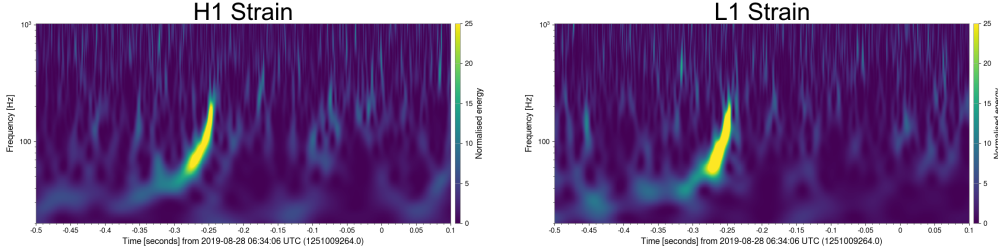

Figure 4: Spectral plot for $GW190828_{0}63405$

the blip glitch in figure 3, due to the characteristic chirp of a gravitational in spiral.

While the majority of glitches can be easy to dismiss, as they have Signal-to-Noise ratios in the hundreds to thousands, it is the quietest that present the largest problem. Those that have Signal-to-Noise ratios (SNR's) on the same scale as true mergers, between 10 and 30, have near identical properties to IMBH events.

Due to their similarity with short duration mergers, there exists the potential that the curious deficit of IMBH merger events could be due to incorrect labelling as glitches. This report will progress toward a search for glitch-like IMBH mergers that may help to place limits on the number of known high-mass events.

1.3 Moving forward

In order to further filter out glitch events, especially those quieter ones that mimic IMBH mergers, it would be best to create a template model that accurately represents these glitch events. Particularly, it should have near identical physical characteristics, and should respond to signal processing in much the same way as the ones encountered within LIGO data.

With an appropriate model created, a search through LIGO data using the same methods that have found gravitational mergers should be able to locate glitch events. Running this search against data that contains both known glitch and merger events should be an excellent test of theory, and allow any fine tuning to more accurately determine the nature of each detection event.

Finally, by comparing and contrasting each of these searches between mul-

tiple detector data, we should be able to determine if any glitches (or mergers) are true glitch events, or misconstrued IMBH mergers. Extending this search further into unknown data should also allow the potential discovery of new IMBH events, and potentially a better understanding into how to accurately model and remove these glitch events.

To do this, the Python PyCBC package will be used inside a Jupyter notebook running on an external LIGO server. This should expose all required LIGO data and computational power to completely achieve the goals laid out above.

2 Theory

2.1 Constructing a Glitch

Before attempting to construct a glitch template, it is prudent to list the known properties of glitches found within LIGO data:

- Merger similarity:

  Glitches respond very similarly to mergers when matched filtering for merger templates. They also exhibit similar spectral properties to known merger events, as touched on in section 1.2.

- Duration:

  Glitches are very short duration, typically on the order of tenths of a second. The specific glitches this paper focuses on, blip glitches, are time-symmetric, unlike merger events that tend to have an initial chirp.

- Loudness:

  Glitches can vary from near-undetectable, to completely overwhelming, with an almost continuous distribution between the two. Any given glitch can have any given loudness, with no obvious relation.

In order to address the first point and ensure our glitch template has similar spectral features and properties to a merger event, we will first start with a merger template as shown in figure 5. While this does give us the

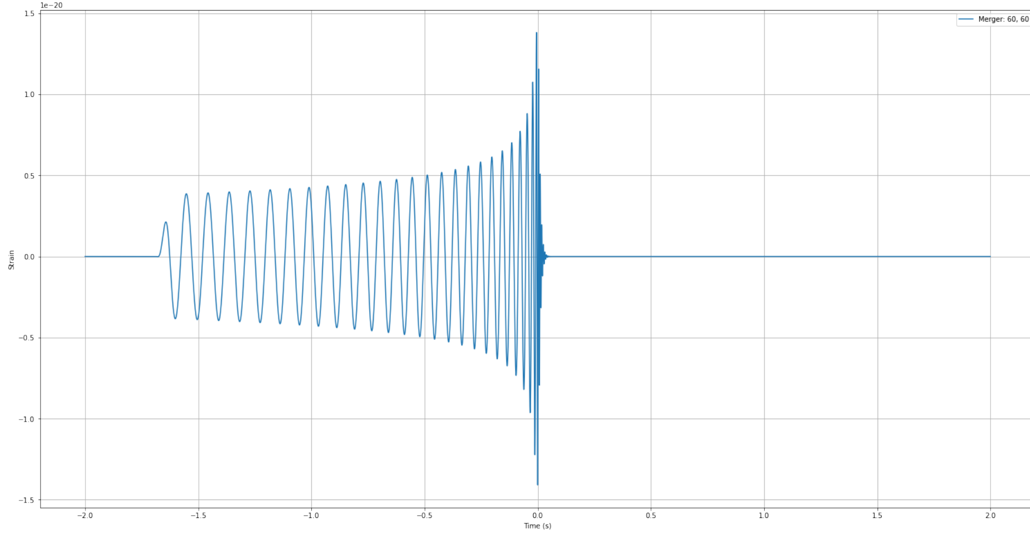

Figure 5: Merger between two $60M_{\odot}$ black holes

characteristic spectrum we desire, with most of the energy contained in lower frequencies, this does come with the side-effect of introducing the characteristic merger chirp into our template.

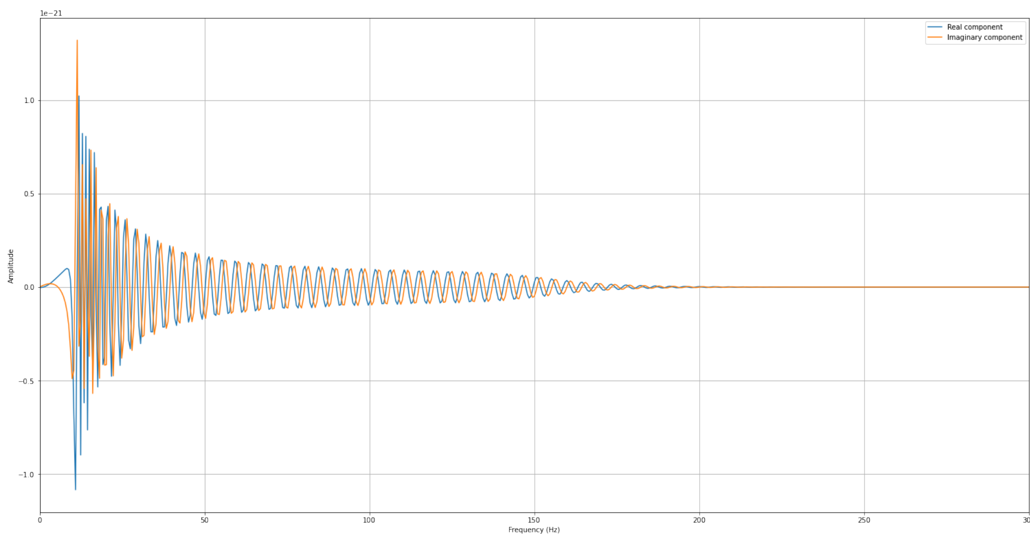

Figure 6: Fourier transform of the merger

As there are very few things that can be done here without removing required information, The template will then be converted to a frequency series by way of Fourier transform as shown in figure 6. This representation encodes each frequency of a waveform as a complex number, where the ar-

gument is the phase of each frequency, and the magnitude is its amplitude. This representation thus allows us to address the second point above.

Standard form

$$z = a + bj \tag{1}$$

$$a = \mathrm{Re}(z) \tag{2}$$

$$b = \mathrm{Im}(z) \tag{3}$$

Polar form

$$z = r(\cos(\theta) + j\sin(\theta)) = re^{j\pi\theta} \tag{4}$$

$$r = \mathrm{mod}(z) = \sqrt{a^2 + b^2} \tag{5}$$

$$\theta = \arg(z) = \arctan(\frac{b}{a}) \tag{6}$$

One way of representing short duration is to say that all frequency information is in phase. As the phases of each individual sinusoidal become aligned, so too does their central peaks, causing constructive interference around the centre and destructive interference elsewhere. As we know that phase information for each frequency is the argument of each complex number, a useful next step would be setting this to zero without affecting the modulus (and subsequently amplitude) for each frequency.

From 6, we can see an easy way of achieving this is setting b, or the imaginary part, to zero. To retain the amplitude information, 5 Shows that $r^2 = a^2 + b^2$, and so a, or the real part, must be set to the modulus. This is, conveniently enough, what the numpy.abs() function does, the output of such shown in figure 7.

Finally, using an Inverse Fourier transform to return to the time domain, we should see that our template now occurs almost exclusively at $t = 0$, as shown in figure 8. As the Inverse Fourier Transform expects a sequence of complex numbers, care should be taken to avoid completely removing the imaginary part in the step above. As numpy.abs() automatically does this, the glitch frequency series had to be recast using numpy.astype("complex-128"), which converts each number to a complex double floating point value (in essence, appending $0j$ to what would otherwise be a sequence of reals).

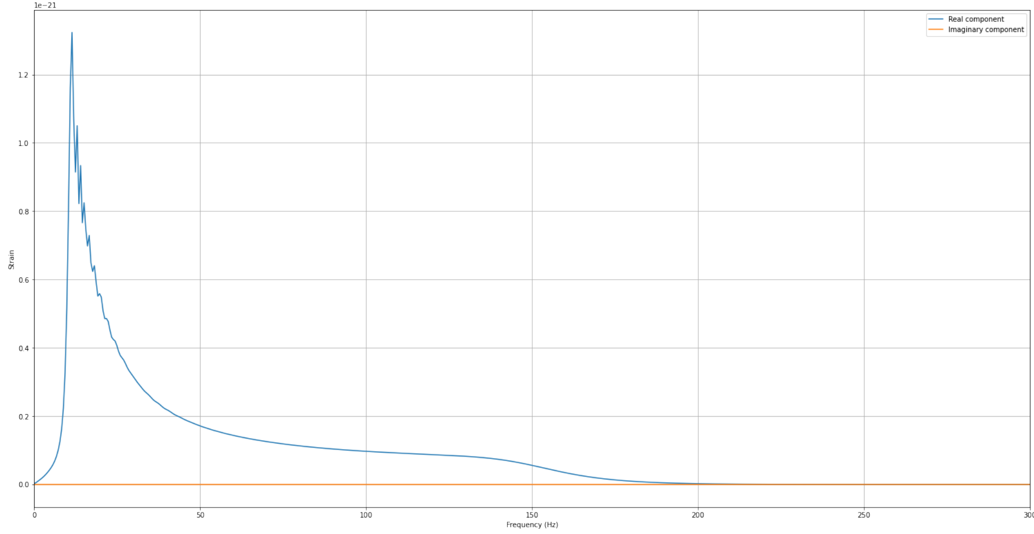

Figure 7: All phase information removed

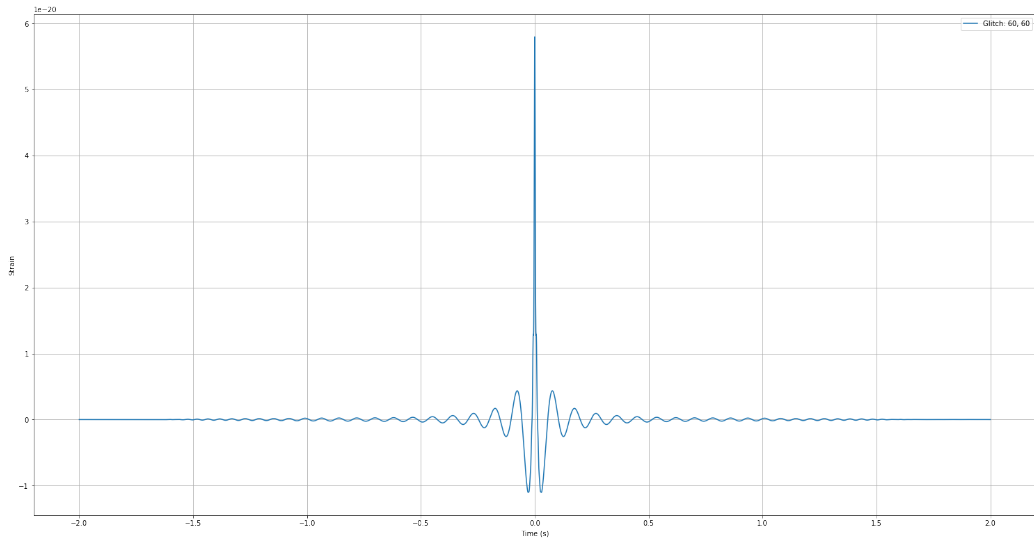

Figure 8: Inverse Fourier transformed into the time domain

There is an animation of the merger to glitch conversion hosted here as part of this project's Github Repository [9].

To assess how similar our glitch model and merger model are, we will use the PyCBC.filter.match() function to compute their similarity. This function takes two templates and yields two numbers, ϵ and ϕ. ϵ is a measure between 0 and 1 of their correlation, where 0 is completely dissimilar and 1 is completely identical, and ϕ is the time offset between the two signals

required to obtain the match. As we are only concerned with how correlated the two signals are at this stage, we can discard ϕ.

To see how the similarity, or ϵ, between glitch and merger varies as a function of mass, we can create a bank of template mergers between two equal mass black holes across a range of masses, and a bank of glitches from those same mergers. While it would not be difficult to use unequal mass templates (such as a glitch formed from a $30M_{\odot}$-$50M_{\odot}$ merger), the equal mass templates are more than appropriate for our needs. Figure 9 shows the result of this operation, where the glitches and templates were generated with symmetric masses between 10 and $300M_{\odot}$. The z axis, which shows ϵ, is also represented proportionally with a colour scale.

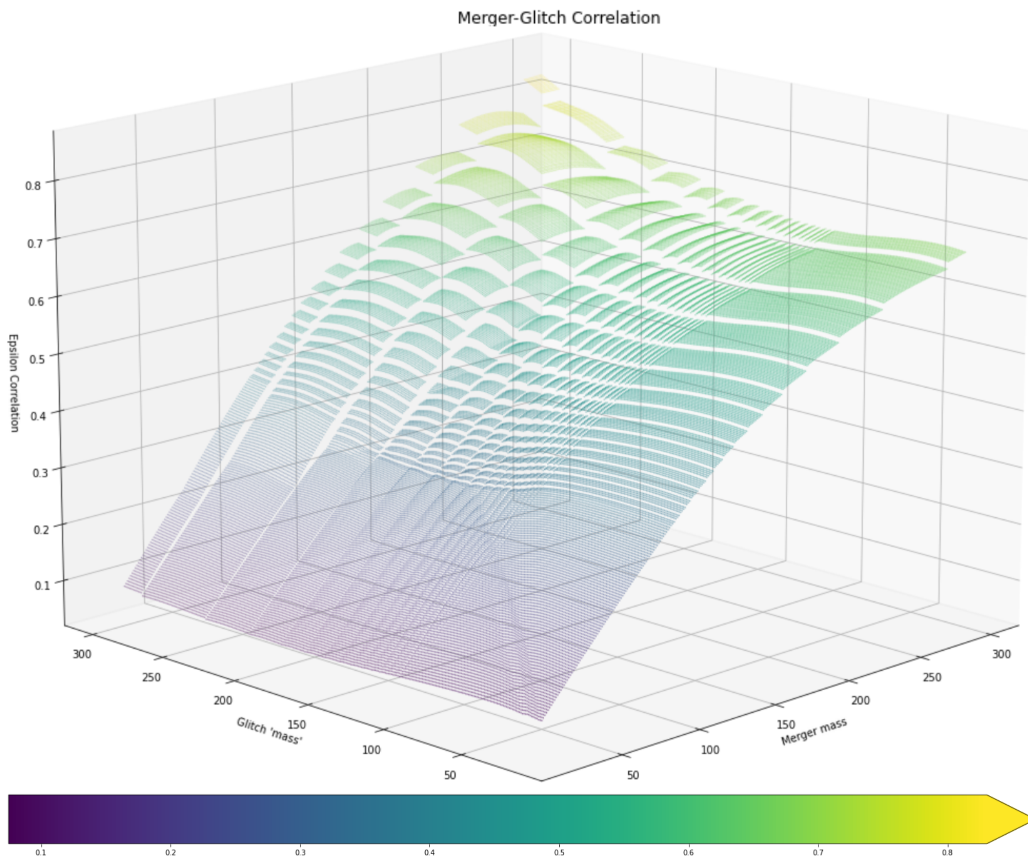

Figure 9: Epsilon correlation (ϵ) between a bank of Glitches and Mergers

We can see that, for low mass merger events, the value of ϵ does not vary strongly as the mass of each glitch increases. This also shows that low mass mergers do not look particularly like glitch events, as ϵ does not rise above

0.2 until the symmetric merger mass is above $50M_{\odot}$, which is already more massive than all mergers observed except GW190521 [1].

The short discontinuities in the graph is an artefact of the computation required to calculate ϵ. The likely culprit is the frequency cutoff chosen when generating the templates, for this particular computation, only information above 10Hz was retained. This would explain why the graph is continuous at low masses, as these mergers contain a lot of information in the 100-300Hz area, while discontinuous at High masses, which occur mostly within the 1-30 Hz regime and thus are missing some of their frequency information.

2.2 Matched filtering

Matched filtering [2] is the main method by which the bulk of this search is performed. This tool is particularly powerful for identifying a known signal within data that contains Gaussian noise, as it is *mathematically provable* [find source maybe?] to be the optimum linear filter. As such, it underpins a lot of work in RADAR and similar subsystems, as they too require filtering known data from noise. The two deceptively simple equations that describe it's working are given below:

$$\rho = \frac{1}{\sigma} \int \frac{d(f)h^*(f)}{S(f)} df \tag{7}$$

$$\sigma^2 = \int \frac{h(f)h^*(f)}{S(f)} df \tag{8}$$

The output of the matched filter function is the signal-to-noise (SNR) ratio for a given template h against data d, represented by ρ in equation 7. The σ term given is the auto-correlation of the template, and is used to normalise the SNR output.

We can see in equation 8 that we multiply the template with its complex conjugate. This operation yields the amplitude squared of the template, with the imaginary portion collapsing to zero, an operation which can be demonstrated with little effort.

$$z = a + bj \tag{9}$$

$$z^* = a - bj \tag{10}$$

$$zz^* = (a + bj)(a - bj) \tag{11}$$

$$zz^* = a^2 - abj + abj - b^2j^2 \tag{12}$$

$$zz^* = a^2 + b^2 = \text{mod}(z)^2 \tag{13}$$

Having obtained the amplitude squared of the template, we divide through by the spectral density of the data. This has the effect of reducing any frequency values in the template that are not present in the data, which if integrated over all frequencies gives the correlation of the template and itself squared.

We then perform a very similar operation with the data and template, using the conjugate of the template as we've already computed it. This causes shared frequency content between the data and template to be retained prominently, while those that aren't shared are diminished. The following division by the spectral density of data evens out regular frequencies found, resulting in spikes for each frequency proportional to the strength of those frequencies present in the template.

By integrating over all frequencies, the relative correlation of the template and data at that point is returned, proportional to the amount of the template in the data. Dividing this by the auto-correlation of the template normalises the filter, such that $\rho = 1$ is equal noise and template content, and any values above that represent a louder template signal found.

A side effect of this operation is that $\rho = 1$ is also one standard deviation of noise, as noise is Gaussian in nature, and so the value of ρ is identical to the standard deviation of the probability of the template occurring in the data by random noise fluctuation; that is, a template with SNR 8.9 has a $1 - \text{erf}(\frac{8.9}{\sqrt{2}}) = 5.58467 * 10^{-17}\%$ chance of being due to random noise fluctuations.

This method of matched filtering with our glitch model stands in contrast to the typical tools used, most prominently the omicron scan, as seen in figure 2. Omicron scanning is agnostic to data context, quite unlike matched filtering with it's specified template searching.

Omicron scanning operates on a wavelet-like basis, whereby the entire data is whitened, and individual tiles formed from the data are overlaid on top of each-other. These tiles vary between large in frequency domain and small in

time, and vice versa, and combine to give an overview of the spectral content of a segment of data. Where multiple of these tiles overlap, the data has deviated from a standard Gaussian, and the spectral shape of this event is shown [11].

It should be useful then, moving forward, to compare the results seen in omicron scans with those found by glitch matched filter searches. While they may not provide identical results, it should still pose as a secondary affirmation of found events.

3 Methodology

3.1 The Search

Now that we have a model to generate glitch events, and a handle of the methods that we can use to search for them, we can combine the two into a set of python scripts to perform a full search.

3.1.1 Data collection

To first begin, the data to perform a search on needed to be obtained. Initial testing when developing the code function used an hour long segment at GPS time 1244473218 (2019-06-13, 15:00), while the full search documented in the results used a 3 hour long segment starting at GPS time 1242442818 (2019-05-21, 03:00).

Data was collected from both Hanford and Livingston detectors, though could easily be extended to include Virgo and others. To ease computational time, the data was down-sampled from its native $16384Hz$ sampling time to $4096Hz$. This data was then separated into smaller chunks of length $512s$ with $32s$ padding either side. As the matched filter requires the template and data to be of equal length, this was a happy medium between reducing the number of matched filters that needed to be computed, and reducing the length of the templates (and subsequently their memory usage). To complete the requirements for the matched filter function, the spectral density for each chunk of data was computed, as shown in figure 10.

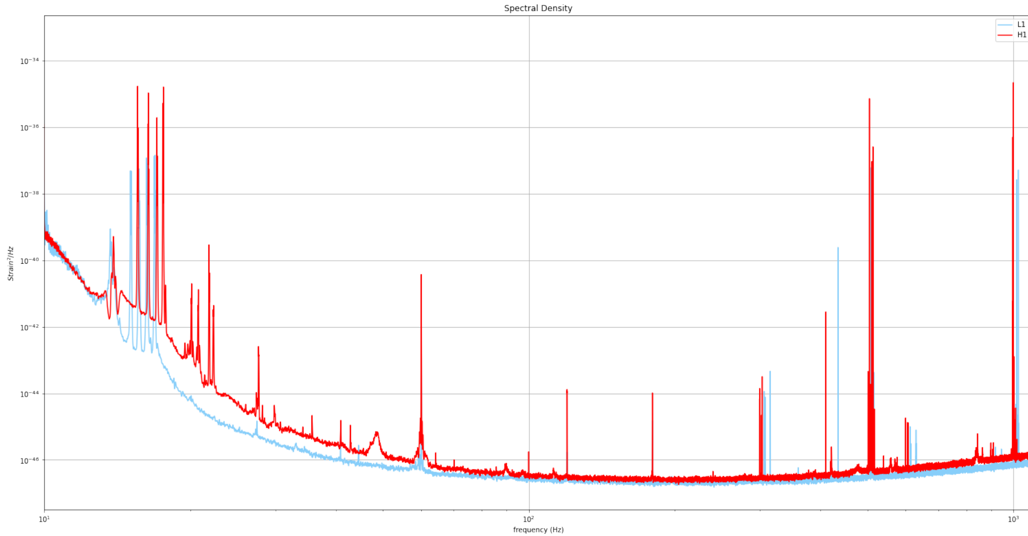

Figure 10: Logarithmic plot of the Spectral density of the two detectors. Note the different characteristic frequencies that occur between the two.

3.1.2 Template Generation

Following this, an entire bank of template glitches and mergers needed to be created. While these can be as numerous as desired, the results in this paper were collected by creating equal-mass templates between $20M_{\odot}$ and $300M_{\odot}$ in $10M_{\odot}$ intervals, for a total of 58. While higher mass templates, and a greater mass resolution between them, could have been used, this made for an appropriate middle ground between computational speed and breadth of search. Each of the templates created had a length of $576s$ ($512s + 32s$ padding either side) and a sampling rate of $4096Hz$ to match each data chunk.

It was imperative that the length of each template was specified before performing a cyclic time shift operation. This operation was used to align the peak of each template with $t = 0$ by wrapping the entire template around its time length. If additional time was appended after this, the wrapped template would be discontinuous at $t = 0$, causing filtering errors, as the matched filter process assumes that all data and templates are continuous. This has the effect of a secondary detection echo occurring when computing the SNR, as the wrapped data is partway through the template, rather than neatly at the end of each template. An example of this artefact is shown here in figure 11, where a secondary peak is detected after a time proportional to the duration of the initial template.

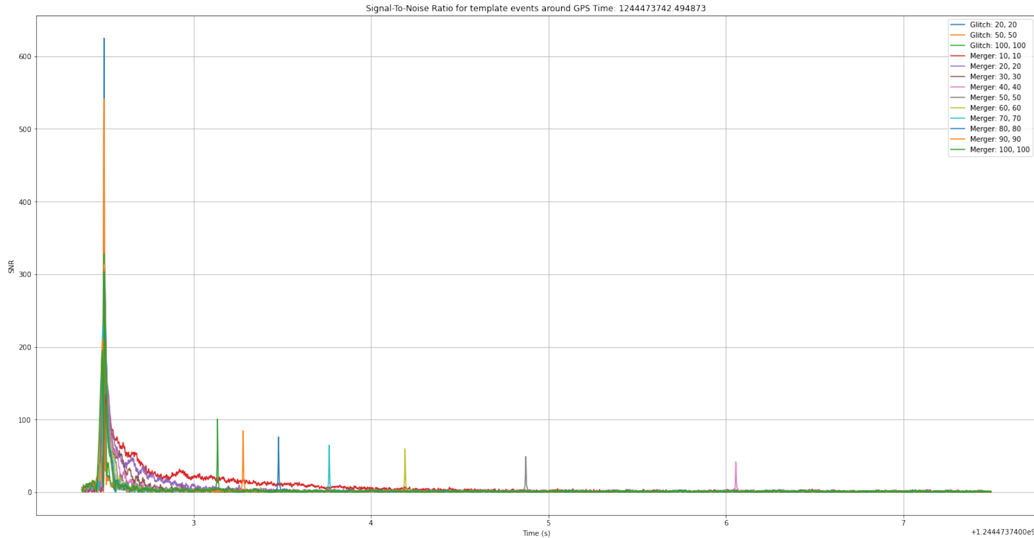

Figure 11: Secondary SNR echo due to incorrect template shifting and resizing. The secondary detection peaks can be seen to occur after the primary.

3.1.3 Signal Processing

For each $576s$ chunk of data, the Signal-to-Noise ratio was computed for each of the template events. As this would have taken a long time to do in series a multiprocessing pool [10] was utilised instead, with each of the 8 workers given a template at a time. While more workers could have been used, this would have slightly increased the overhead per worker, and would have consumed more resources on the shared server.

With 928 SNR segments of length $576s$ found, all of the points where the SNR dropped below a signal threshold were discarded, so that only significant peaks remained. A typical SNR of 8 is chosen for LIGO searches, but as the standard deviation for Gaussian noise is ± 1, an event that would have had a raw SNR of 8 could feasibly register as 7 with noise included, hence SNR 7 was the cutoff for this search.

For each of the peaks found for a given template, all peaks within a certain time threshold were compared, such that only the loudest within a $5s$ boundary was reported. This eliminates echoing artefacts, and is significantly under the expected detection time of a gravitational event. Once each of the most significant peaks for each of the templates is found, they are compiled into an extended array, and our signal processing stages are complete.

3.2 Compiling results

3.2.1 Event Detection

With an array of each SNR peaks found within the data, compiling them into a coherent list of probable events is the next, and arguably most important, step. To ensure computational efficiency from this point onward, the peak array is sorted chronologically. This makes the task of determining which templates align much simpler, as any two neighbours greater than $5s$ apart mark the boundary between one event and the next.

It should be ensured between these two steps that events are sorted when other parts of the search expect them to be, as this could (and did) cause cascading errors in follow up calculations.

By splitting this array into sub-arrays, each containing only coincident SNR peak events (those that occur within a $5s$ boundary of each other), we can directly compare each template to identify which was responsible for this event. The simplest approach is to assume that the template with the loudest SNR is most likely responsible for this event.

As we have only the coincident templates for each event, we can also show which detector data was triggered for this event. This, coupled with identifying a glitch or merger as the most probable culprit, provides an important step in quickly determining which, if any, event requires further study.

While not directly required, a very useful metric to calculate and show at this stage is G/M ratio, or the SNR of the loudest glitch for this event divided by the SNR of the loudest merger. This serves as a very quick indication of how "glitchy" an event actually is, with values close to 1 representing a possibility that random noise could have pushed this event one way or the other. Alongside this, $G - M$ offset, or the time delay between the peak of the loudest glitch and loudest merger, can also be shown, though this is less important for collecting results.

For example, supposing that a 20 glitch and 130 merger had an SNR of 14.5 and 10 respectively, separated by $0.5s$. This would be identified as a single $20M_{\odot}$ glitch, with a G/M ratio of 1.45. As these individual peaks are separated by more than ± 1, we can be reasonably confident in saying this event is a glitch.

Once the above steps have all been completed, it remains only to distribute them across a table, as shown in both extended and summary tables. Then, with all events categorised and laid out logically, any anomalies or objects of further study can be identified.

3.2.2 Graphical Output

While the search is technically complete, as the table of results contains any information needed, it does not necessarily aid an understanding of the distribution of events. To that end, every template peak would then be plotted on a graph, with colour representing detector, and shape showing glitch or merger, as shown in the results figure 16.

This graph would thus make coincident template defections obvious, and would also show the approximate glitch frequency in an easy to understand format.

4 Results

For the bulk of this section, we will be referring to the table of results returned by the search script that can be found at the end of this document, with exception given to a zoomed in figure comparison below. Secondarily, all quoted SNR's (in table or otherwise) have an implicit error of ± 1 due to Gaussian noise while error in reported time is assumed to be $\pm \frac{1}{4096} s$ due to the sampling time. Calculations using these values also have these implicit errors built in. Finally, the results and search script can be found in this project's Github Repo [9].

As demonstrated in the theoretical segment of this paper, the glitch model looks very similar to known glitches, providing a strong incentive that this model would be effective. We also see a clear correlation of these events with those found by the omicron scan, with a few exceptions whose peak frequency was in the kilohertz regime, as seen in figure 12. This is mostly due to our focus on IMBH mergers, whose frequency content lies in the single to tens of Hertz regime.

As listed in our summarised table of results in section 6.3, we found 61 unique events, of which 59 were initially labelled as glitches. The first event,

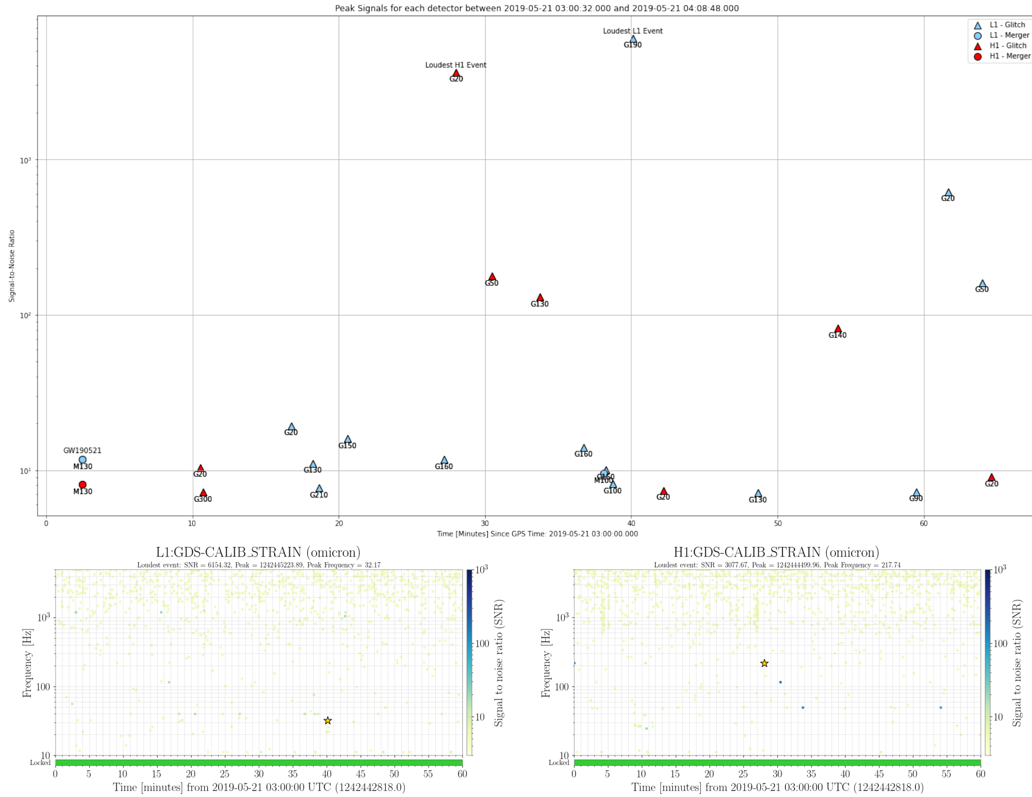

Figure 12: Comparison for one hour of data of Search script output and LIGO omicron scan

GW190521, was correctly identified as a merger, despite using only a few non-specific merger templates as a control for the search. This demonstrates that, while our glitch model is similar to the glitches identified in LIGO, it does not trigger falsely for real mergers.

The second of the two events (event 13) that triggered as a merger took place at 03:38:07, 36 minutes after GW190521, was identified as a $100M_{\odot}$-$100M_{\odot}$ merger. While it would be incredibly unlikely to have identified the second ever IMBH merger in history within such a short time of the first, it still warranted a further investigation as part of the search pipeline. This event was triggered only in the L1 data, and with a G/M ratio of 0.955 and peak SNR of 9.57 ± 1, hence within the range that Gaussian noise can affect. To that end, it is easily explained as a misidentified glitch in a noisy segment of data.

Event 50 also stood out as an event of interest in our table. This event was identified as a glitch, with peak SNR 2212.58 ± 1 and G/M 1.142 (and thus

well outside the range of Gaussian noise). This would have been nothing of note, if not for the detection trigger in both the L1 and H1 data. By noting its position on the full table of results (row 2206), we can see that this event has two main parts, one that occurred in Hanford at 05:03:51 with SNR $\approx$ 2200, and one in Livingston at 05:03:53 with SNR $\approx$ 7.4. As the disparity between detector SNR was so high, this event is clearly a rare near-coincidental set of two glitches, and reducing the time threshold in the search from $5s$ to $2s$ would have avoided this issue.

With the results obtained, it seems likely that this glitch model is accurate, despite this report only being a pilot study. With a higher mass resolution and larger bank of glitches (as mentioned previously) a much more refined glitch search could be carried out. A logical secondary step after this would be a search for glitches using this model, removal of those identified glitches, and then a secondary search afterwards on the cleaned data, with a much higher confidence that an event found after is a real merger.

5 Conclusion

In conclusion, it has been demonstrated that a glitch model is not only possible, but surprisingly effective. With this, while we may not currently be able to explain what causes glitches, we can definitely model and filter them. Similar to our initial understanding of gravity, we understand how it looks, but not yet what causes it.

Without actively trying to ensure its safety, the glitch model did not misidentify GW190521, only identifying known glitches as glitches. While one glitch was misidentified as a merger, it seems far more useful to generate an occasional false positive, than false negative.

While no additional IMBH mergers have been identified in the limited scope of this paper, it would require only computational time and very minimal effort to extend the search over (potentially) the entirety of the O3 run and beyond, which should provide a definitive answer to the question as to IMBH mergers.

Following this, a larger suite of glitched templates with varying properties could be created and searched for as part of the main LIGO search pipeline. While such a model can be extended to even higher masses, it should be

noted that, from figure 9, we can see that glitches generated via this model and real mergers rapidly converge. By performing a quick calculation, we can note that an $\epsilon = 0.95$ reached for symmetric masses over 800.

6 Appendix

6.1 Bibliography

[1] R. Abbott et al. "GW190521: A Binary Black Hole Merger with a Total Mass of 150 $M_\odot$". In: *Physical Review Letters* 125.10 (2020). DOI: 10.1103/PhysRevLett.125.101102.

[2] J. C. Bancroft. "Introduction to matched filters". In: *CREWES Research Report* 14 (2002), pp. 46.1–46.8.

[3] Wikimedia Commons. *Remnants of massive single stars as a function of initial mass and initial content of elements heavier than Helium (metallicity)*. 2013. URL: https://commons.wikimedia.org/wiki/File:Remnants_of_single_massive_stars.svg.

[4] Wikimedia Commons. *Supernovae types as a function of initial mass and initial content of elements heavier than helium (metallicity)*. 2013. URL: https://en.wikipedia.org/wiki/File:Supernovae_as_initial_mass-metallicity.svg.

[5] Gary S Fraley. "Supernovae explosions induced by pair-production instability". In: *Astrophysics and Space Science* 2 (1 Aug. 1968). ISSN: 1572-946X. DOI: 10.1007/BF00651498. URL: https://doi.org/10.1007/BF00651498.

[6] C. L. Fryer, S. E. Woosley, and A. Heger. "Pair-Instability Supernovae, Gravity Waves, and Gamma-Ray Transients". In: *The Astrophysical Journal* 550.1 (Mar. 2001), pp. 372–382. DOI: 10.1086/319719. URL: https://doi.org/10.1086/319719.

[7] *Gravity spy, list of glitches*. URL: https://www.zooniverse.org/projects/zooniverse/gravity-spy/collections.

[8] Mélanie Habouzit et al. "On the number density of 'direct collapse' black hole seeds". In: *Monthly Notices of the Royal Astronomical Society* 463.1 (Aug. 2016), pp. 529–540. ISSN: 0035-8711. DOI: 10.1093/mnras/stw1924. eprint: https://academic.oup.com/mnras/article-pdf/463/1/529/18469622/stw1924.pdf. URL: https://doi.org/10.1093/mnras/stw1924.

[9] Jack Lloyd-Walters. *Github Repository For this project*. 2021. URL: https://github.com/SK1Y101/GWProject/tree/main.

[10] *MultiProcessing Pool Documentation.* URL: https://docs.python.org/3/library/multiprocessing.html#multiprocessing.pool.Pool.

[11] Florent Robinet et al. *Omicron: a tool to characterize transient noise in gravitational-wave detectors.* 2020. arXiv: 2007.11374 [astro-ph.IM].

6.2 List of Figures

1 Supernovae types, and remnant object, given initial star mass and metallicity. credit: [3] [4] 1

2 Omicron scan of the Hanford and Livingston detectors, demonstrating the frequency of glitch events 3

3 Sample Glitch event in the Livingston Detector. The colour scale is the normalised energy for this time range 3

4 Spectral plot for $GW190828_063405$ 4

5 Merger between two $60M_\odot$ black holes 6

6 Fourier transform of the merger 6

7 All phase information removed 8

8 Inverse Fourier transformed into the time domain 8

9 Epsilon correlation (ϵ) between a bank of Glitches and Mergers 9

10 Logarithmic plot of the Spectral density of the two detectors. Note the different characteristic frequencies that occur between the two. 13

11 Secondary SNR echo due to incorrect template shifting and resizing. The secondary detection peaks can be seen to occur after the primary. 14

12 Comparison for one hour of data of Search script output and LIGO omicron scan 17

13 Histogram showing Glitch SNR divided by Merger SNR for each detection event. 25

14 Histogram showing Signal-to-Noise Ratios for every template trigger. 25

15 Histogram showing the number of triggers per template. . . . 26

16 Graphical representation of loudest events triggered for each detector, separated by glitch, merger and detector. 26

6.3 Results

	Summary of Gravitational Signals between 2019/05/21, 03 : 00 and 2019/05/21, 06 : 25, Signal Threshold SNR >= 7.0.					
value unit	Event	Detector	Tempate name	G/M Ratio #	UTC Time ISO string	Signal SNR #
0	Event 1 - Merger	H1, L1	Merger: 130, 130	G/M Ratio: 0.876	2019-05-21 03:02:29.432	11.764142725717386
1	Event 2 - Glitch	H1	Glitch: 20, 20		2019-05-21 03:10:32.179	10.41947421949494
2	Event 3 - Glitch	H1	Glitch: 300, 300	G/M Ratio: 1.02	2019-05-21 03:10:42.591	7.244996515937195
3	Event 4 - Glitch	L1	Glitch: 20, 20	G/M Ratio: 1.726	2019-05-21 03:16:45.594	19.369217359927497
4	Event 5 - Glitch	L1	Glitch: 130, 130	G/M Ratio: 1.183	2019-05-21 03:18:14.778	11.088583339387
5	Event 6 - Glitch	L1	Glitch: 210, 210		2019-05-21 03:18:39.473	7.712564812912004
6	Event 7 - Glitch	L1	Glitch: 150, 150	G/M Ratio: 1.217	2019-05-21 03:20:36.281	15.94708626328262
7	Event 8 - Glitch	L1	Glitch: 160, 160	G/M Ratio: 1.078	2019-05-21 03:27:11.357	11.742573709126612
8	Event 9 - Glitch	H1	Glitch: 20, 20	G/M Ratio: 1.865	2019-05-21 03:28:01.965	3643.5364462830853
9	Event 10 - Glitch	H1	Glitch: 50, 50	G/M Ratio: 1.531	2019-05-21 03:30:28.119	177.76593104528942
10	Event 11 - Glitch	H1	Glitch: 130, 130	G/M Ratio: 1.113	2019-05-21 03:33:46.332	130.3740752879017
11	Event 12 - Glitch	L1	Glitch: 160, 160	G/M Ratio: 1.151	2019-05-21 03:36:44.967	14.043560529935538
12	Event 13 - Merger	L1	Merger: 100, 100	G/M Ratio: 0.955	2019-05-21 03:38:07.260	9.579593968482643
13	Event 14 - Glitch	L1	Glitch: 150, 150	G/M Ratio: 1.215	2019-05-21 03:38:16.167	10.079510582311729
14	Event 15 - Glitch	L1	Glitch: 100, 100	G/M Ratio: 1.068	2019-05-21 03:38:44.878	8.2083938974249
15	Event 16 - Glitch	L1	Glitch: 190, 190	G/M Ratio: 1.072	2019-05-21 03:40:05.899	6034.256211671037
16	Event 17 - Glitch	H1	Glitch: 20, 20		2019-05-21 03:42:11.619	7.435904918918023
17	Event 18 - Glitch	L1	Glitch: 130, 130		2019-05-21 03:48:39.914	7.17226848896819
18	Event 19 - Glitch	H1	Glitch: 140, 140	G/M Ratio: 1.132	2019-05-21 03:54:07.267	82.26821904587578
19	Event 20 - Glitch	L1	Glitch: 90, 90		2019-05-21 03:59:29.786	7.263838439540498
20	Event 21 - Glitch	L1	Glitch: 20, 20	G/M Ratio: 2.454	2019-05-21 04:01:40.566	619.4686347884222
21	Event 22 - Glitch	L1	Glitch: 50, 50	G/M Ratio: 1.681	2019-05-21 04:03:59.863	161.26635501382947
22	Event 23 - Glitch	H1	Glitch: 20, 20		2019-05-21 04:04:38.510	9.0630836994015
23	Event 24 - Glitch	L1	Glitch: 290, 290	G/M Ratio: 1.036	2019-05-21 04:13:01.835	318.1359565294674
24	Event 25 - Glitch	L1	Glitch: 270, 270	G/M Ratio: 1.062	2019-05-21 04:17:04.964	13.17909676401721
25	Event 26 - Glitch	H1	Glitch: 80, 80	G/M Ratio: 1.35	2019-05-21 04:21:32.904	3259.294441092481
26	Event 27 - Glitch	H1	Glitch: 30, 30	G/M Ratio: 1.962	2019-05-21 04:34:48.398	53.842667926856066
27	Event 28 - Glitch	H1	Glitch: 20, 20	G/M Ratio: 1.995	2019-05-21 04:35:06.503	206.65090363804728
28	Event 29 - Glitch	L1	Glitch: 20, 20	G/M Ratio: 2.271	2019-05-21 04:35:50.441	1573.7050412838698
29	Event 30 - Glitch	L1	Glitch: 170, 170		2019-05-21 04:36:36.048	7.874631478020997
30	Event 31 - Glitch	H1	Glitch: 40, 40	G/M Ratio: 1.112	2019-05-21 04:36:46.098	16.97511849597848
31	Event 32 - Glitch	H1	Glitch: 40, 40	G/M Ratio: 1.629	2019-05-21 04:37:10.978	133.94052581842575
32	Event 33 - Glitch	H1	Glitch: 50, 50	G/M Ratio: 1.6	2019-05-21 04:40:30.680	211.21825388392318
33	Event 34 - Glitch	L1	Glitch: 120, 120	G/M Ratio: 1.095	2019-05-21 04:48:24.261	15.519938214096918
34	Event 35 - Glitch	L1	Glitch: 190, 190	G/M Ratio: 1.061	2019-05-21 04:51:13.445	436.52591713201264
35	Event 36 - Glitch	L1	Glitch: 20, 20		2019-05-21 04:56:13.202	7.520083716322648
36	Event 37 - Glitch	L1	Glitch: 200, 200	G/M Ratio: 1.08	2019-05-21 04:57:07.480	10.22783222719123
37	Event 38 - Glitch	L1	Glitch: 140, 140	G/M Ratio: 1.187	2019-05-21 05:01:16.551	9.581785583644624
38	Event 39 - Glitch	L1	Glitch: 210, 210	G/M Ratio: 1.062	2019-05-21 05:02:24.835	130.04983793933826
39	Event 40 - Glitch	L1	Glitch: 130, 130	G/M Ratio: 1.23	2019-05-21 05:02:50.138	12.776878218748388
40	Event 41 - Glitch	L1	Glitch: 40, 40	G/M Ratio: 1.093	2019-05-21 05:08:14.056	124.07857546524777
41	Event 42 - Glitch	L1	Glitch: 30, 30	G/M Ratio: 1.17	2019-05-21 05:09:47.383	316.4275708994588
42	Event 43 - Glitch	H1	Glitch: 20, 20	G/M Ratio: 2.071	2019-05-21 05:13:11.171	986.7696131410048

	Summary of Gravitational Signals between 2019/05/21, 03 : 00 and 2019/05/21, 06 : 25, Signal Threshold SNR >= 7.0.					
value	Event	Detector	Tempate name	G/M Ratio	UTC Time	Signal SNR
unit				#	ISO string	#
43	Event 44 - Glitch	H1	Glitch: 30, 30	G/M Ratio: 1.563	2019-05-21 05:16:18.768	33.11058772120468
44	Event 45 - Glitch	L1	Glitch: 130, 130	G/M Ratio: 1.164	2019-05-21 05:16:47.122	9.343690221403033
45	Event 46 - Glitch	H1	Glitch: 80, 80	G/M Ratio: 1.248	2019-05-21 05:18:33.077	563.4179658184607
46	Event 47 - Glitch	L1	Glitch: 170, 170	G/M Ratio: 1.184	2019-05-21 05:19:13.816	14.724972773504406
47	Event 48 - Glitch	L1	Glitch: 120, 120		2019-05-21 05:24:29.668	7.643574052110319
48	Event 49 - Glitch	H1	Glitch: 20, 20	G/M Ratio: 1.977	2019-05-21 05:32:42.454	51.49588986991503
49	Event 50 - Glitch	H1, L1	Glitch: 140, 140	G/M Ratio: 1.142	2019-05-21 05:33:51.322	2212.5781367497966
50	Event 51 - Glitch	H1	Glitch: 20, 20		2019-05-21 05:36:28.943	8.659741129755359
51	Event 52 - Glitch	L1	Glitch: 180, 180		2019-05-21 05:45:17.938	7.508272550763041
52	Event 53 - Glitch	L1	Glitch: 140, 140	G/M Ratio: 1.131	2019-05-21 05:47:37.162	10.958745381569596
53	Event 54 - Glitch	L1	Glitch: 160, 160	G/M Ratio: 1.141	2019-05-21 05:54:16.771	54.01640288978776
54	Event 55 - Glitch	L1	Glitch: 200, 200	G/M Ratio: 1.099	2019-05-21 05:55:21.702	16.706859018369798
55	Event 56 - Glitch	L1	Glitch: 240, 240	G/M Ratio: 1.045	2019-05-21 05:58:02.374	607.2484061727234
56	Event 57 - Glitch	H1	Glitch: 220, 220	G/M Ratio: 1.048	2019-05-21 06:01:23.991	9.418672590842208
57	Event 58 - Glitch	L1	Glitch: 160, 160	G/M Ratio: 1.088	2019-05-21 06:08:40.854	10.662594673750185
58	Event 59 - Glitch	L1	Glitch: 90, 90		2019-05-21 06:13:32.956	8.35676072152142
59	Event 60 - Glitch	L1	Glitch: 160, 160	G/M Ratio: 1.157	2019-05-21 06:14:03.177	39.80657219651998
60	Event 61 - Glitch	H1	Glitch: 20, 20	G/M Ratio: 1.283	2019-05-21 06:21:25.875	27.887348369579332

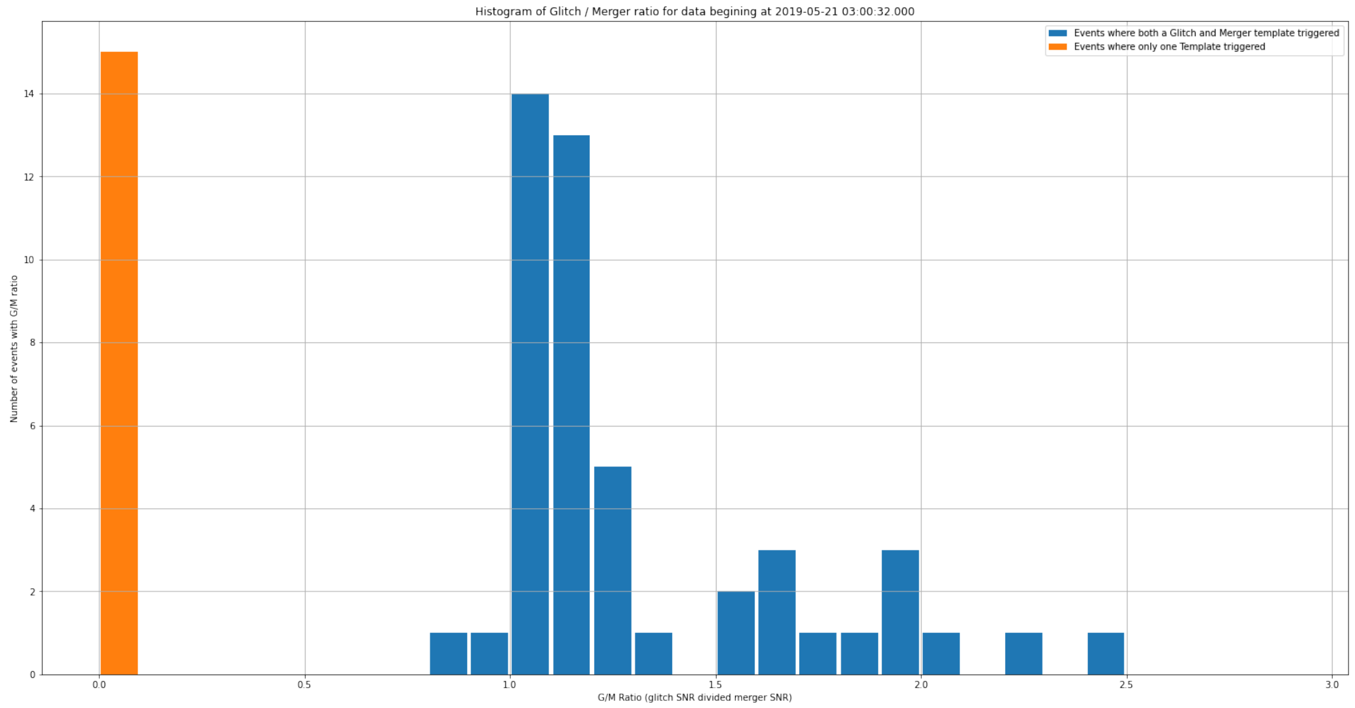

Figure 13: Histogram showing Glitch SNR divided by Merger SNR for each detection event.

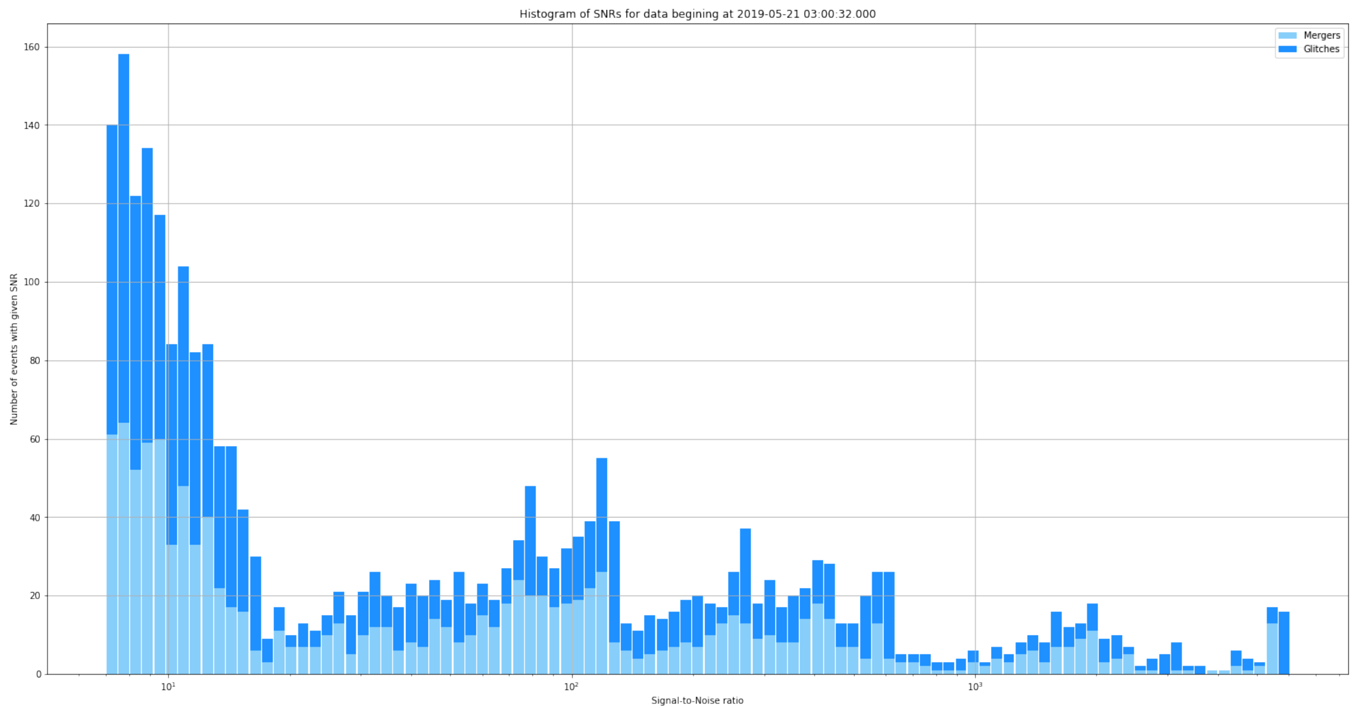

Figure 14: Histogram showing Signal-to-Noise Ratios for every template trigger.

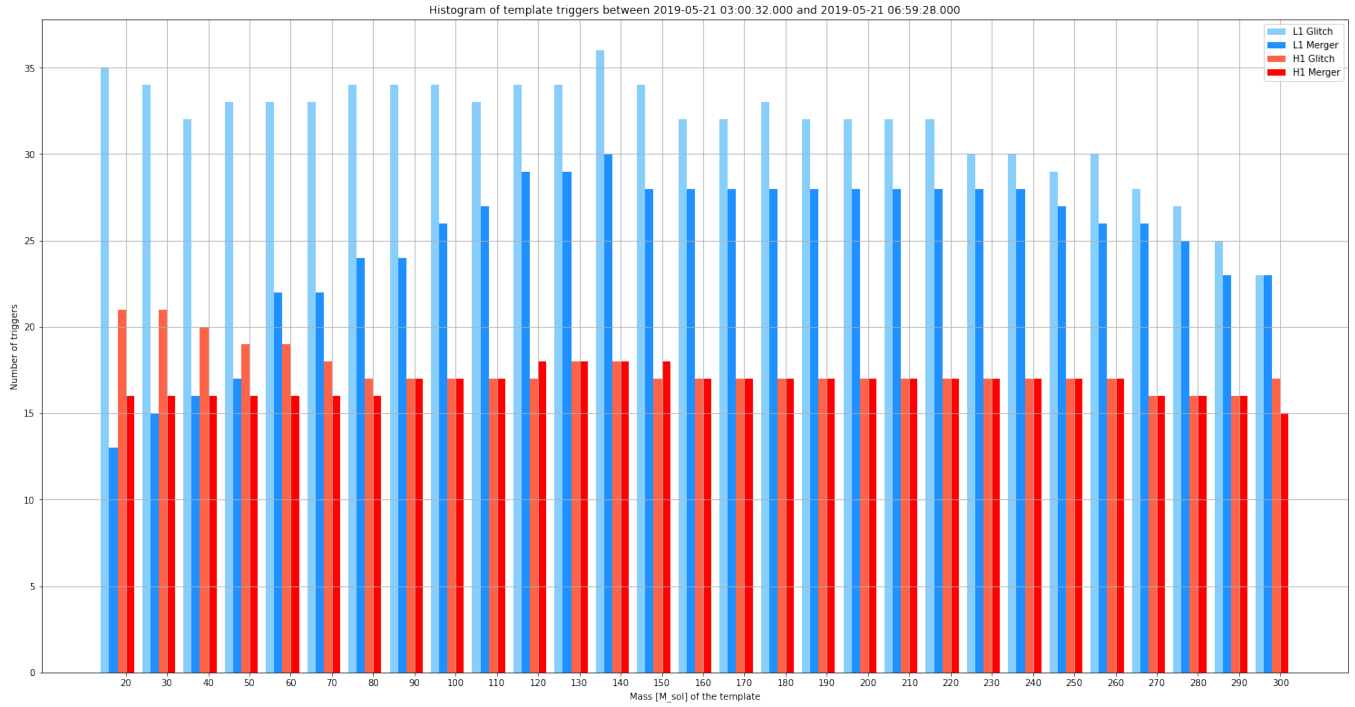

Figure 15: Histogram showing the number of triggers per template.

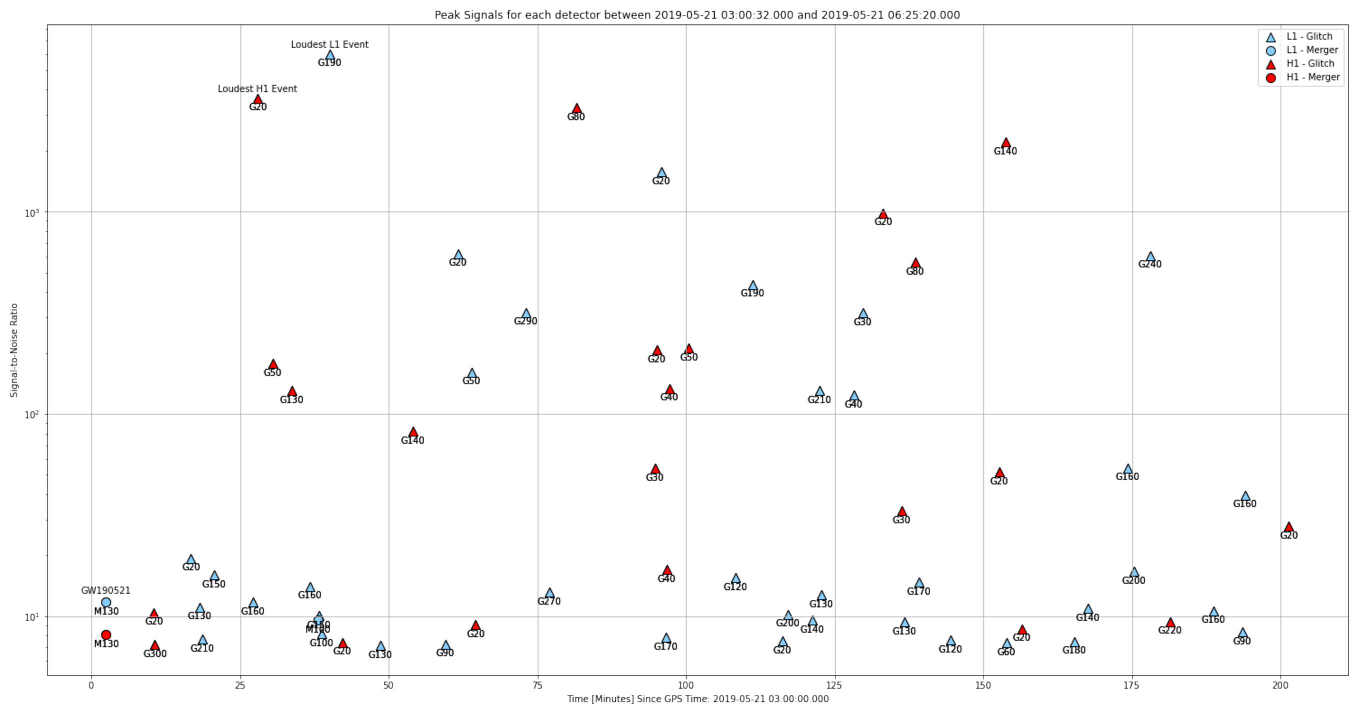

Figure 16: Graphical representation of loudest events triggered for each detector, separated by glitch, merger and detector.

6.4 Search Program

6.4.1 External Functions

```
1  #import the pycbc specific modules
2  from pycbc.filter import resample_to_delta_t, matchedfilter,
       matched_filter
3  from pycbc.psd import interpolate, inverse_spectrum_truncation
4  from pycbc.waveform import taper_timeseries, get_td_waveform
5  from pycbc import types, frame
6
7  #Import all of the general modules we require
8  from timeit import timeit, default_timer
9  from multiprocessing import Pool
10 from astropy.time import Time
11 import pandas as pd
12 import numpy as np
13 import pylab
14
15 #set the environment variable
16 %env LIGO_DATAFIND_SERVER=ldr.ldas.cit:80
17
18 #Define the class that will handle parrallel computing
19 class ParrallelJobHandler():
20     #initialisation
21     def __init__(self, processes = 8):
22         #define the number of pool workeds this will use
23         self.processes = processes
24
25     #execute a function with an array of inputs in parrallel
26     def runJob(self, function, *inputs, unpack=False):
27         funcname = function.__name__
28         print()
29         #ensure that nested inputs are adequately dealt with
30         inputs = expandArray(*inputs, unpack=unpack)
31         #create a counter to show how many processes are
               remaining
32         numleft, maxnum = 0, 0
33         #create the pool
34         with Pool(self.processes) as p:
35             #split the inputs among the workers
36             r = p.starmap_async(function, inputs)
37             #check the job is still running
38             while not r.ready():
39                 #check if a job has been completed
40                 if (numleft != r._number_left):
41                     #update our counter
42                     numleft = r._number_left
```

```
43                      maxnum = max(maxnum, numleft)
44                      #show a nice display to the user
45                      showProgress(funcname, maxnum - numleft,
                            maxnum)
46          showProgress(funcname, maxnum, maxnum, overwrite=False)
47          #store the results of the job internally
48          self.lastResult = r.get()
49          #return the results
50          return self.lastResult
51
52      #execute a parrallel job, and report the time taken
53      def timedJob(self, function, *inputs, unpack=False):
54          #create the timeit, and execute
55          t = timeit(lambda: self.runJob(function, *inputs, unpack
                =unpack), number = 1)
56          #once complete, show the runtime
57          print(" - {} completed in {:.2f}s".format(function.
                __name__, t))
58          #and return the output of the job
59          return self.lastResult
60
61      #execute a single function and report the time taken
62      def timedFunc(self, function, *inputs):
63          def execute(self, function, *inputs):
64              self.lastResult = function(*inputs)
65          #create the timeit, and execute
66          t = timeit(lambda: execute(self, function, *inputs),
                number = 1)
67          #once complete, show the runtime
68          print(" - {} completed in {:.2f}s".format(function.
                __name__, t))
69          #and return the output of the job
70          return self.lastResult
71
72  #function that takes a tupple of inputs, and returns an array of
        unique combinations of the inputs
73  def expandArray(*inputs, useLCM=True, unpack=False):
74      #compute the number of unique values we need based on the
            lengths of all lists in the inputs
75      b = [1, 1] + list([len(x) for x in inputs if isinstance(x, (
            tuple, list))])
76      #compute the total combinations using product or LCM
77      if useLCM:
78          b = int(np.lcm.reduce(b))
79      else:
80          b = int(np.product(b))
81
82      #create a new array formed of the inputs
83      mixed = np.array([[y[x%len(y)] if isinstance(y, (tuple, list
```

```
            )) else y for y in inputs] for x in range(b)], dtype="
            object")
84      #if the array contained additional tupples that need to be
            unpacked, this will handle them
85      mixed = np.array([[z for y in x for z in (y if isinstance(y,
             (list, tuple)) else (y,))] for x in mixed], dtype="
            object")
86      #try to return the sorted array
87      try:
88          #sort the array by column
89          sort = mixed[np.lexsort(np.transpose(mixed)[::-1])]
90      #else just return the mixed
91      except:
92          sort = mixed
93      return [tuple(x) for x in sort]
94
95  #Function to print a nice loading sequence
96  def showProgress(operation, current, total, length=100,
        overwrite=True):
97      #ensure the inputs are valid
98      if total:
99          #convert to a fraction
100         progress = max(0, float(current / total))
101         #setup the text to print
102         done = int(round(progress * length))
103         todo = length - done
104         text = "␣-␣{}␣\t␣[{}]␣{:.1f}%".format(operation, "#"*
                done + "-"*todo, progress*100)
105         #check if we have reached the end of the bar
106         if overwrite:
107             #print the text as is
108             print(text, end="\r")
109         else:
110             #otherwise print the text without the return
                    sequence
111             print(text)
112
113 #ensure that data provided is in a plottable format
114 def toPlottable(data):
115     #ensure the data is in a dictionary format
116     if not isinstance(data, dict):
117         #check if the data is nested timeseries
118         if isinstance(data[0], (types.FrequencySeries, types.
                TimeSeries)):
119             #try to fetch plotname from meta data
120             try:
121                 return dict([(x.plotName, x) for x in data])
122             #otherwise, give a default name
123             except:
```

```
124                 return dict([("Waveform", x) for x in data])
125         #if the data is not
126         else:
127             #try and fetch it's plotname
128             try:
129                 return {data.plotName: data}
130             except:
131                 return {"waveform": data}
132     #return the nicely formatted data
133     return data
134
135 def subPlot(data, title="", xlab="", ylab="", xlim=None, ylim=
    None, sharex=True, sharey=True, figsize=(10,10),
136             grid=True, asLogLog=False, asQTransform=True,
                notePeak=False, savePlotName="", maxplots=1000):
137     #ensure the data provided is acceptable
138     data = toPlottable(data)
139     maxplots = min(maxplots, len(data))
140
141     #fetch the figure and axes
142     fig = pylab.figure(figsize=(figsize[0], maxplots*figsize[1])
        )
143     gs = fig.add_gridspec(maxplots, hspace=0)
144     axes = gs.subplots(sharex=sharex, sharey=sharey)
145
146     #itterate through the dictionary
147     y = 0
148     for x in data:
149         #check if we have a frequency series
150         if isinstance(data[x], types.FrequencySeries):
151             xplot = data[x].sample_frequencies
152             #apply the xlabel if not done so already
153             if not xlab:
154                 xlab = "Frequency (Hz)"
155         #else check for timeseries
156         elif isinstance(data[x], types.TimeSeries):
157             xplot = data[x].sample_times
158             if not xlab:
159                 xlab = "Time (s)"
160
161         #check if we are plotting as a Log-Log graph
162         if asLogLog:
163             axes[y].loglog(xplot, data[x], label=x)
164         if asQTransform:
165             t, f, p = data[x].whiten(4, 4).qtransform(.001,
                logfsteps=100,qrange=(8, 8),frange=(10, 512))
166             axes[y].pcolormesh(t, f, p**0.5, vmin=1, vmax=6,
                shading="auto")
167             axes[y].set_yscale('log')
```

```
168                 xlab='Time␣(s)'
169                 ylab='Frequency␣(Hz)'
170             else:
171                 axes[y].plot(xplot, data[x], label=x)
172 
173             #check if we are meant to note the location of the peak
                    on this axis
174             if False:#notePeak:
175                 posx = data[x].numpy().argmax()
176                 posy = xplot[posx]
177                 axes[y].text(posx, posy, "{}:␣{},␣{}".format(x, posx
                        , posy), fontsize="small")
178             #increment to the next axes
179             y = (y+1)%maxplots
180 
181         #set the title
182         axes[0].set_title(title)
183 
184         #show the plot, and label the outer
185         for ax in axes:
186             #set the axes labels
187             ax.set_xlabel(xlab)
188             ax.set_ylabel(ylab)
189             #try to add a legend if possible
190             try:
191                 ax.legend(loc="best")
192             except:
193                 pass
194             #check limits
195             if xlim:
196                 ax.set_xlim(xlim)
197             if ylim:
198                 ax.set_ylim(ylim)
199             #add a grid
200             if grid:
201                 ax.grid()
202             #ensure that the labels are only on the outermost
203             ax.label_outer()
204 
205         fig.show()
206 
207         #if a save name has been supplied
208         if savePlotName:
209             #remove any fileextensions if given, and add '.png'
                    instead
210             savename = "{}.png".format(str(savePlotName).split(".")
                    [0])
211             #save the figure
212             fig.savefig(savename, bbox_inches='tight')
```

```
213
214 #plot the data
215 def plotData(data, title="", xlab="", ylab="", xlim=None, ylim=
        None, figsize=(10,10),
216                 grid=True, asLogLog=False, asQTransform=False,
                    notePeak=False, savePlotName=""):
217     #create the plot
218     fig = pylab.figure(figsize=figsize)
219     ax = fig.add_axes((0, 0, 1, 1))
220
221     #ensure the data provided is acceptable
222     data = toPlottable(data)
223     #itterate through the provided data and plot it
224     for x in data:
225         #check if we have a frequency series
226         if isinstance(data[x], types.FrequencySeries):
227             xplot = data[x].sample_frequencies
228             #apply the xlabel if not done so already
229             if not xlab:
230                 xlab = "Frequency (Hz)"
231         #else check for timeseries
232         else:
233             xplot = data[x].sample_times
234             if not xlab:
235                 xlab = "Time (s)"
236
237         #check if we are plotting as a Log-Log graph
238         if asLogLog:
239             ax.loglog(xplot, data[x], label=x)
240         if asQTransform:
241             t, f, p = data[x].whiten(4, 4).qtransform(.001,
                    logfsteps=100,qrange=(8, 8),frange=(10, 512))
242             ax.pcolormesh(t, f, p**0.5, vmin=1, vmax=6, shading=
                    "auto")
243             ax.set_yscale('log')
244             xlab='Time (s)'
245             ylab='Frequency (Hz)'
246         else:
247             ax.plot(xplot, data[x], label=x)
248
249         #check if we are meant to note the location of the peak
                on this axis
250         if notePeak:
251             posx = data[x].numpy().argmax()
252             posy = xplot[posx]
253             ax.text(posx, posy, "{}: {}, {}".format(x, posx,
                    posy), fontsize="small")
254
255     #set the title
```

```
256        ax.set_title(title)
257        #set the axes labels
258        ax.set_xlabel(xlab)
259        ax.set_ylabel(ylab)
260        #try to add a legend if possible
261        try:
262            ax.legend(loc="best")
263        except:
264            pass
265        #add a grid
266        if grid:
267            ax.grid()
268        #check limits
269        if xlim:
270            ax.set_xlim(xlim)
271        if ylim:
272            ax.set_ylim(ylim)
273        #show the plot
274        fig.show()
275
276        #if a save name has been supplied
277        if savePlotName:
278            #remove any fileextensions if given, and add '.png'
                 instead
279            savename = "{}.png".format(str(savePlotName).split(".")
                 [0])
280            #save the figure
281            fig.savefig(savename, bbox_inches='tight')
282
283
284    #plot data on a nice pandas dataframe.
285    def asTable(*inputs, title="", latexName=""):
286        #create our dataframe
287        df = pd.DataFrame()
288        #create some empty arrays that will hold the headers, etc
289        headers, titles, units = [], [], []
290        #iterate through the supplied inputs to fetch each thing to
             plot
291        for x in range(len(inputs)):
292            #ensure we have data in the correct form
293            if not (isinstance(inputs[x], dict) and ("data" in
                 inputs[x].keys())):
294                #if the input is not a dictionary that contains a
                     data value, then go to the nex input
295                continue
296            thisdict = inputs[x]
297            #if we've been given a header
298            headers = np.append(headers, [thisdict["header"] if "
                 header" in thisdict else ""])
```

```
299            #or a units column
300            units = np.append(units, [thisdict["unit"] if "unit" in
                   thisdict else ""])
301            #add our title to the title array
302            titles = np.append(titles, title)
303            #add the data to the table
304            try:
305                df[x] = thisdict["data"]
306            except:
307                df[x] = thisdict["data"].flatten()
308        #set the column titles
309        df.columns = pd.MultiIndex.from_arrays([titles, headers,
               units], names=[" ", "value", "unit"])
310        #ensure we see all rows
311        pd.set_option('display.max_rows', None)
312        #and show the display
313        display(df)
314        #if we were given a latex savename
315        if latexName:
316            #save as a latex table
317            df.to_latex("{}.tex".format(latexName.replace(".tex",""))
                   ))
318
319    #function to grab our metadata from an object
320    def fetchMeta(obj):
321        class meta:
322            def __init__(self, plotName=None, detectorName=None):
323                self.plotName = plotName
324                self.detectorName = detectorName
325        #return the new metadata only object
326        try:
327            try:
328                return meta(obj.plotName, obj.detectorName)
329            except:
330                return meta(obj.plotName)
331        except:
332            return meta()
333
334    #function to add metadata to an object
335    def addMeta(new, old, addtoname=""):
336        new.plotName = old.plotName
337        new.detectorName = old.detectorName
338        if addtoname:
339            new.plotName="{} {}".format(new.plotName, addtoname)
340        return new
341
342    #return the absolute value of a waveform
343    def toABS(waveform):
344        return abs(waveform).astype("complex128")
```

```
345
346 #convert a time (orarray of times) to UTC string
347 def toUTC(times):
348     #if the times given is not itterable
349     if isinstance(times, (str, float, int)):
350         try:
351             #try to compute the time
352             return str(Time(Time(float(times), format="gps"),
                    format="iso", scale="utc"))
353         except:
354             return times
355     #otherwise, assume it is itterable
356     return [toUTC(x) for x in times]
357
358 #convert a frequencyseries to a time one
359 def toTime(waveform):
360     if not isinstance(waveform, types.TimeSeries):
361         meta = fetchMeta(waveform)
362         waveform = waveform.to_timeseries()
363         return addMeta(waveform, meta)
364     return waveform
365
366 #convert a time to a freq
367 def toFreq(waveform):
368     if not isinstance(waveform, types.FrequencySeries):
369         meta = fetchMeta(waveform)
370         waveform = waveform.to_frequencyseries()
371         return addMeta(waveform, meta)
372     return waveform
373
374 #function to normalise a waveform
375 def normalise(data, psd=None, lowfreq=10):
376     #fetch the sigma
377     data_sigma = matchedfilter.sigma(data, psd=psd,
            low_frequency_cutoff=lowfreq)
378     #divide the data by the normalisation
379     norm = data / data_sigma
380     #add the matadata, return it
381     return addMeta(norm, data, "normalised")
382
383 #compute the signal-to-noise ratio of a template against a data
        stream
384 def fetchSNR(template, data, psd=None, lowfreq=10, padding=8):
385     #fetch the metadata
386     template.detectorName = data.detectorName
387     meta = fetchMeta(template)
388     #ensure the template is in the time domain (this will be a
            glitch)
389     if not isinstance(template, types.TimeSeries):
```

```
390             template = template.to_timeseries(data.delta_t)
391         #compute the signal-to-noise
392         snr = matched_filter(template, data, psd=psd,
                low_frequency_cutoff=lowfreq)
393         #crop out the padding from the snr, and ensure we have the
                absolute
394         snr = abs(snr.crop(padding, padding))
395         return addMeta(snr, meta, "SNR")
396
397 #search for peaks using a faster method I hope
398 def searchPeaks(snr, signalThreshold = 8, timeThreshold = 100):
399     #convert the snr to a numpy array
400     snr = snr.numpy()
401     #find all the peaks
402     peaks = np.where(snr >= signalThreshold)[0]
403     #check we found any peaks
404     #we ignore all single index peaks as errors
405     if len(peaks) > 1:
406         #compute the boundary region around each signal,
                ensureing we only have unique peaks
407         idxs = 1 + np.where(peaks[1:] - peaks[:-1] >
                timeThreshold)[0]
408         idxs = np.unique(np.concatenate(( [peaks[0]], peaks[idxs
                ], [peaks[-1]] )), axis=0)
409         #compute the exact location of each peak within the
                boundaries
410         for x in range(len(idxs)-1):
411             locations = [int(idxs[x]) + snr[idxs[x] : idxs[x
                    +1]].argmax() for x in range(len(idxs)-1)]
412         #and return the locations of the peaks
413         return locations
414     #return empty if we did not find any
415     return []
416
417 #search through an array of SNR's, comparing those from
        different sources to find the one most likely to have caused
        a signal
418 def searchEvents(templates, peaks, snr, timeThreshold = 1):
419     #an empty array of events
420     events = np.zeros((1, 5))
421     snrsection = []
422     #itterate through all peak times
423     for x in range(len(peaks)):
424         #fetch the current information
425         tempname, snname, pk, sn = templates[x].plotName, snr[x
                ].detectorName, peaks[x], snr[x]
426         #find the SNR and exact time for each event, and add it
                to the events array
427         for y in pk:
```

```
428                 #fetch the utc time for this event
429                 thistime = float(sn.sample_times[y])
430                 #stack each peak onto the event array
431                 events = np.vstack(( events, [thistime, toUTC(
                        thistime), float(sn[y]), tempname, snname] ))
432                 #fetch the SNR around the signal, and append with
                        metadata. also ensure our SNR indices within the
                        bounds of the SNR
433                 section = sn[max(0, int(y - timeThreshold/sn.delta_t
                        )) : min(int(y + timeThreshold/sn.delta_t), len(
                        sn))]
434                 snrsection.append(addMeta(section, sn, "around_{}".
                        format(thistime)))
435         #remove the zeros array we added to the start
436         events = np.delete(events, 0, axis=0)
437         #fetch the chronological sorting
438         sortidx = events[:,0].argsort()
439         #and return the array, by sorting in place. Note: The SNR
                array needs to be sorted slightly differently to preserve
                 the PyCBC-ness of it
440         return events[sortidx], [snrsection[x] for x in sortidx]
441
442     #a function that will return an array whose contents includes a
            specified substring
443     def fetchStringArray(array, string):
444         #attempt to find the substring in place
445         try:
446             return array[np.where(np.char.find(array, string)>=0)
                    [0]]
447         except:
448             #otherwise, ensure we are searching a string array
449             return array[np.where(np.char.find(np.char.array(array),
                    bytes(string, 'utf-8'))>=0)[0]]
450
451     #convert an array of raw events into a neatly analysed array of
            events
452     #by finding all clusters of merger events that occur within the
            signal threshold of eachother.
453     def findClusters(events, timeThreshold=1):
454         '''start as:    time, utctime, snr, name, detector
455            end as:      event no. time, utctime, snr, name, detector
                 '''
456         #create some shorthand for each column index
457         cols = 1
458         evtcol, timecol, utccol, snrcol, namecol, detcol = np.arange
                (cols+events.shape[1])
459         #create an array of empty columns
460         empty_columns = np.full((len(events), cols), "", dtype="
                object")
```

```
461     #and add them to our array
462     events = np.c_[empty_columns, events]
463     summary = np.zeros((1, events.shape[1]))
464     #compute the boundary region around each signal
465     peaktimes = events[:, cols].astype("float64")
466     idxs = 1 + np.where(peaktimes[1:] - peaktimes[:-1] >
            timeThreshold)[0]
467     #add the first and last indices, and invert the order
468     idxs = np.concatenate(( [0], idxs, [len(events)] ))[::-1]
469     #itterate through the indices
470     for x in range(len(idxs)-1):
471         #fetch the detections events to compare
472         tocompare = events[idxs[x+1]:idxs[x]]
473         #seperate them into glitches and mergers
474         glitches = fetchStringArray(tocompare, "Glitch")
475         mergers = fetchStringArray(tocompare, "Merger")
476
477         #if we had a glitch:
478         if len(glitches) > 0:
479             #the loudest will be first
480             loudglitch = glitches[0]
481             #unless we have more than one
482             if len(glitches) > 1:
483                 loudglitch = glitches[np.array(glitches[:,snrcol
                        ], dtype="float").argmax()]
484                 #find the merger in the events and add a tag
485                 events[int(np.where((events == loudglitch).all(
                        axis=1))[0][0])][evtcol] = "Loudest_Glitch"
486
487         #if we had a merger:
488         if len(mergers) > 0:
489             #the loudest will be first
490             loudmerger = mergers[0]
491             #unless we have more than one
492             if len(mergers) > 1:
493                 loudmerger = mergers[np.array(mergers[:,snrcol],
                        dtype="float").argmax()]
494                 #find the merger in the events and add a tag
495                 events[int(np.where((events == loudmerger).all(
                        axis=1))[0][0])][evtcol] = "Loudest_Merger"
496
497         #fetch the loudest event
498         loudevent = tocompare[0]
499         if len(tocompare) > 1:
500             loudevent = tocompare[np.array(tocompare[:,snrcol],
                    dtype="float").argmax()]
501
502         #and if we had both a merger and glitch signal
503         merge_glitch_ratio, merge_glitch_offset = "", ""
```

```
504             if len(mergers) and len(glitches):
505                 merge_glitch_ratio = "G/M Ratio: {}".format(round(float(loudglitch[snrcol]) / float(loudmerger[snrcol]), 3))
506                 merge_glitch_offset = "G-M Offset: {}s".format(round(float(loudglitch[timecol]) - float(loudmerger[timecol]), 3))
507 
508             #insert the event analysis if we had more than one event
509             if len(tocompare) > 1:
510                 events = np.insert(events, idxs[x+1], loudevent, axis=0)
511                 events[idxs[x+1]][timecol] = merge_glitch_ratio
512                 events[idxs[x+1]][utccol] = merge_glitch_offset
513 
514             #add the event number to the column
515             events[idxs[x+1]][evtcol] = "Event {} - {}".format(len(idxs) - x - 1, loudevent[namecol].split(":")[0])
516             #add the detectors to the column (If a singal was seen in multiple detectors, it will be shown here)
517             events[idxs[x+1]][detcol] = ", ".join(np.unique(tocompare[:, detcol]))
518             #add this event to our summary array
519             summary = np.vstack((summary, events[idxs[x+1]]))
520             summary[-1][utccol] = loudevent[utccol]
521             #if we have only a single value, ensure we don't report GPS time in our G/M Ratio Column
522             if len(tocompare)==1:
523                 summary[-1][timecol] = ""
524             #insert an empty space before the summary
525             events = np.insert(events, idxs[x+1], np.full(loudevent.shape, "", dtype="object"), axis=0)
526         #remove the initial zeros from our summary array
527         summary = np.delete(summary, 0, axis=0)[::-1]
528         #return the analysed events
529         return events, summary
530 
531 #fetch the spectral density of a waveform
532 def fetchPSD(data, lowfreq=10, psdtime=4):
533     meta = fetchMeta(data)
534     #ensure we have a timeseries
535     if not isinstance(data, types.TimeSeries):
536         data = data.to_timeseries()
537     #compute psd
538     psd = data.psd(psdtime)
539     #interpolate
540     psd = interpolate(psd, data.delta_f)
541     #now do my favourite function
542     psd = inverse_spectrum_truncation(psd, int(data.sample_rate
```

```
            * psdtime), low_frequency_cutoff=lowfreq)
543     #apply metadata and return
544     return addMeta(psd, meta, "spectral density")
545
546 #function to fetch data from a source
547 def fetchData(name, channel, gps_time, length=8, delta_t
        =1.0/4096):
548     #fetch the data
549     ts = frame.query_and_read_frame(name, channel, gps_time,
            gps_time+length)
550     #resample to delta_t
551     ts = resample_to_delta_t(ts, delta_t)
552     #add our metadata
553     ts.plotName="{}, {}".format(name, gps_time)
554     ts.detectorName = channel.split(":")[0]
555     return ts
556
557 #split data into equally spaced subsections
558 def splitData(data, length=64, padding=8):
559     #fetch the metadata
560     meta = fetchMeta(data)
561     #calculate the number of splitable sections, such that some
            padding is removed from the end, and each is a defined
            length
562     sections = np.floor((data.duration - 2*padding) / length)
563     #convert the time durations to indices
564     padding /= data.delta_t
565     length /= data.delta_t
566     #split the array, with overlapping sections, adding metadata
             to each
567     return [ addMeta(data[int(x * length) : int((x+1) * length +
             2 * padding)], meta, "+{}s".format(x*length*data.delta_t
            )) for x in range(int(sections)) ]
568
569 #combine two operations into a single function call
570 def fetchSplitData(name, channel, gps_time, chunks=1, sublength
        =128, padding=8, delta_t=1.0/4096):
571     #fetch the data
572     data = fetchData(name, channel, gps_time, chunks*sublength +
             2*padding, delta_t)
573     #and return the split data
574     return splitData(data, sublength, padding)
575
576 #create a merger using PyCBC
577 def createMerger(mass1, mass2, distance=100, delta_t=1.0/4096,
        tlen=None, lowfreq=10, approximant="SEOBNRv4_opt"):
578     #create the waveform
579     waveform = get_td_waveform(approximant=approximant, mass1=
            mass1, mass2=mass2,
```

```
580                                     delta_t=delta_t , f_lower=lowfreq ,
                                            distance=distance)[0]
581     #taper the waveform
582     waveform = taper_timeseries(waveform, "TAPER_START")
583     #resize the template if neccesary
584     if tlen:
585         waveform.resize(int(tlen / delta_t))
586         #shift the data so that the strongest point occurs at t
                =0
587         waveform = waveform.cyclic_time_shift(waveform.
                start_time)
588     #add our metadata
589     waveform.plotName="Merger: {}, {}".format(mass1, mass2)
590     #return the merger
591     return waveform
592 
593 #create a glitch using our model
594 def createGlitch(mass1, mass2, distance=100, delta_t=1.0/4096,
        tlen=None, lowfreq=10, approximant="SEOBNRv4_opt"):
595     #create the waveform
596     waveform = get_td_waveform(approximant=approximant, mass1=
            mass1, mass2=mass2, delta_t=delta_t , f_lower=lowfreq ,
            distance=distance)[0]
597     #taper the waveform
598     waveform = taper_timeseries(waveform, "TAPER_START")
599     #resize the template if neccesary
600     if tlen:
601         waveform.resize(int(tlen / delta_t))
602         #shift the data so that the strongest point occurs at t
                =0
603         waveform = waveform.cyclic_time_shift(waveform.
                start_time)
604     #convert the waveform to a glitch by taking the absolute
605     waveform = toABS(waveform.to_frequencyseries())
606     #add our metadata
607     waveform.plotName="Glitch: {}, {}".format(mass1, mass2)
608     #return the merger
609     return waveform
```

6.4.2 Search Script

```
1 ### Constant setup ###
2 
3 #initialise our parrallel job handler
4 job = ParrallelJobHandler(10)
5 
6 #fetch the detector and gpstime
```

```
7  channel = "L1:GDS-CALIB_STRAIN"
8  frametype = "L1_HOFT_C00"
9  gps_time = 1242442818
10 #Current test period: 1250553618
11 #Start of 2019-05-21 1242432018
12 #IMBH check time: 1242442818
13 #Initial testing time: 1244473218
14 delta_t = 1.0/4096
15
16 #1242442818 GPS TIME CONTAINS GW190521, so hopefully it will
       trigger the detection
17
18 #setup the length values for each item
19 chunks = 24 # 24 = 3hrs
20 padding = 32
21 sublength = 512
22 #16 * 512 ~= 2 hours of data
23
24 ### Data Collection ###
25
26 #fetch the data
27 print("Fetching L1 Data")
28 dataArray = fetchSplitData(frametype, channel, gps_time, chunks,
        sublength, padding, delta_t)
29 print("Fetching H1 Data")
30 dataArray += fetchSplitData(frametype.replace("L1", "H1"),
       channel.replace("L1", "H1"), gps_time, chunks, sublength,
       padding, delta_t)
31
32 #show an example of the data split
33 #print("Showing data")
34 #subPlot(dataArray, figsize=(20, 3), title="Data split into
       smaller chunks", maxplots=2, grid=True)
35
36 #fetch the spectral density of each section of data
37 print("fetching spectral density")
38 psdArray = job.timedJob(fetchPSD, dataArray, 10, 4)
39
40 #show an example of the spectral density
41 print("Showing Spectral density")
42 plotData([psdArray[0], psdArray[chunks]], xlim=(10, 1100),
       figsize=(20, 10), xlab="frequency", ylab='$Strain^2 / Hz$',
       title="Spectral Density", asLogLog=True)
43
44 ### Merger Generation ###
45
46 print("Creating Merger Templates")
47 #create an array of mass pairs for the merger templates
48 minmass = 20
```

```
49 maxmass = 300
50 massstep = 10
51
52 #create an array of mass pairs for the templates
53 massArray = [(x, x) for x in np.arange(minmass, maxmass+massstep
       , massstep)]
54
55 #the length of each template should match our data chunk, with
       some padding
56 template_length = dataArray[0].duration
57
58 #compute the set of mergers
59 mergers = job.timedJob(createMerger, massArray, 100, delta_t,
       template_length, unpack=True)
60
61 #and plot them to show everything works in order
62 print("Plotting Example Mergers")
63 plotData(mergers[0], title="Merger Examples", figsize=(20, 5))
64
65 ### Glitch Generation ###
66
67 #create an array of mass pairs for the glitches
68 print("Creating Glitch Templates")
69 #create an array of mass pairs for the merger templates
70 Gminmass = 20
71 Gmaxmass = 300
72 Gmassstep = 10
73
74 #create an array of mass pairs for the templates
75 GmassArray = [(x, x) for x in np.arange(Gminmass, Gmaxmass+
       Gmassstep, Gmassstep)]
76
77 #compute a set of viable glitches (50, 50) responds best to
       known glitch events
78 glitches = job.timedJob(createGlitch, GmassArray, 100, delta_t,
       template_length)
79
80 #and plot them to show everything works in order
81 print("Plotting Example Glitches")
82 plotData(glitches[0], title="Glitch Examples", figsize=(20, 5),
       xlim=(0, 300))
83
84 ### SNR Search ###
85
86 #compile the template events into an array
87 templates = glitches+mergers
88
89 #setup the constants for the functions
90 lowfreq = 10
```

```
91 signalThreshold = 7  # The SNR must be above this value to count
92 timeThreshold = 5    # Two SNR peaks must be at least this far
       apart (in seconds) to count
93
94 #start out our events array with a run of zeros. We will have to
        remove this afterwards.
95 events = np.zeros((1, 5))
96
97 for x in range(len(dataArray)):
98     #fetch the current chunks of data
99     dataChunk, psdChunk = dataArray[x], psdArray[x]
100    print("Searching: {}".format(dataChunk.plotName))
101    #Load from save file
102    try:
103        evts = np.load("savedData/{}.npy".format(dataChunk.
               plotName))
104        print("\n - Loading SNRs")
105        events = np.vstack((events, evts))
106    #otherwise, compute as neccesary
107    except:
108        print("\n - Computing SNRs")
109        #compute the SNR for each template
110        SNRs = job.timedJob(fetchSNR, templates, dataChunk,
               psdChunk, lowfreq, padding)
111
112        print("\n - Finding Peaks")
113        #find all of the peaks in the SNR
114        peaks = job.timedJob(searchPeaks, SNRs, signalThreshold,
                timeThreshold / delta_t)
115
116        #plotData(SNRs, title="{} - SNR".format(dataChunk.
               plotName))
117        print("\n - Compiling Probable Events")
118        #fetch the events within our timeThreshold
119        evts = job.timedFunc(searchEvents, templates, peaks,
               SNRs, timeThreshold)[0]
120        #save this data for future usage
121        np.save("savedData/{}.npy".format(dataChunk.plotName),
               evts)
122        #concatenate with the previously found events
123        events = np.vstack((events, evts))
124
125    #print a small newline seperator to separate the outputs
           slightly
126    print("\n\n")
127
128    SNRs, peaks = None, None
129
130 #remove the run of zeros we started with
```

```
131 events = np.delete(events, 0, axis=0)
132
133 # We don't have to sort the data normally, but if we are
        splicing data between multiple detectors, this is neccesary
134 if len(dataArray) > chunks:
135     #(if we have more data points than we should if we only
            queried one detector)
136     sortidx = events[:,0].argsort()
137     #sort using the sorting index found above
138     events = events[sortidx]
139     #waves = [waves[x] for x in sortidx]
140
141 #Latex name stuff
142 dateTime = toUTC(gps_time).split(" ")[0][2:].replace("-","")
143 timeLen = "{}hr".format(int(np.floor((chunks*sublength) / 3600))
        )
144
145 #Save the events array for later use if neccesary
146 np.save("{}-{}-events.npy".format(dateTime, timeLen), events)
147
148 ### Tabulate Data ###
149
150 #fetch the times for the title
151 start_time = gps_time + padding
152 end_time = start_time + chunks*sublength
153 #search through the events to find overlaping times
154 analysed_events, summarised_events = findClusters(events,
        timeThreshold)
155
156
157 #Latex name stuff
158 '''runNo = 2
159 dateTime = toUTC(gps_time).split(" ")[0][2:].replace("-","")
160 timeLen = "{}hr".format(int(np.floor((chunks*sublength) / 3600))
        )'''
161
162
163 #split the summary array into a few subarrays
164 evtnum, gmratio, utcTime, value, names, detector = np.hsplit(
        summarised_events, 6)
165 #show the summary data on a table
166 asTable({"header":"Event", "unit":"", "data":evtnum},
167         {"header":"Detector", "unit":"", "data":detector},
168         {"header":"Tempate name", "unit":"", "data":names},
169         {"header":"G/M Ratio", "unit":"", "data":gmratio},
170         {"header":"UTC Time", "unit":"ISO string", "data":
                utcTime},
171         {"header":"Signal SNR", "unit":"#", "data":value},
172         title="Summary of Gravitational Signals between {} and
```

```
            {}, Signal Threshold SNR >= {}".format(toUTC(
            start_time), toUTC(end_time), float(signalThreshold))
            ,
173         latexName="{}-{}-Summary_TimeSorted".format(dateTime,
            timeLen))
174
175
176 #split the summary array into a few subarrays, sorted by SNR
177 evtnum, gmratio, utcTime, value, names, detector = np.hsplit(
        summarised_events[summarised_events[:,3].astype("float64").
        argsort()], 6)
178 #show the summary data on a table
179 asTable({"header":"Event", "unit":"", "data":evtnum},
180         {"header":"Detector", "unit":"", "data":detector},
181         {"header":"Tempate name", "unit":"", "data":names},
182         {"header":"G/M Ratio", "unit":"", "data":gmratio},
183         {"header":"UTC Time", "unit":"ISO string", "data":
            utcTime},
184         {"header":"Signal SNR", "unit":"#", "data":value},
185         title="Summary of Gravitational Signals between {} and
            {}, Signal Threshold SNR >= {}".format(toUTC(
            start_time), toUTC(end_time), float(signalThreshold))
            ,
186         latexName="{}-{}-Summary_SNRSorted".format(dateTime,
            timeLen))
187
188
189 #split the events array into a few subarrays
190 evtnum, time, utcTime, value, names, detector = np.hsplit(
        analysed_events, 6)
191 #show the extended data on a table
192 asTable({"header":"Event", "unit":"", "data":evtnum},
193         {"header":"Detector", "unit":"", "data":detector},
194         {"header":"Tempate name", "unit":"", "data":names},
195         {"header":"GPS Time", "unit":"Time", "data":time},
196         {"header":"UTC Time", "unit":"ISO string", "data":
            utcTime},
197         {"header":"Signal SNR", "unit":"#", "data":value},
198         title="Extended view of Gravitational Signals between {}
             and {}, Signal Threshold SNR >= {}".format(toUTC(
            start_time), toUTC(end_time), float(signalThreshold))
            ,
199         latexName="{}-{}-Full_Results".format(dateTime, timeLen)
            )
```