A Data Mining Approach to Signal Processing in Laser Interferometer Gravitational-Wave Observatory (LIGO) Fscan Data

by

Thomas Robert Brooks Harris

A thesis submitted in partial fulfillment of the requirements
for graduation with Honors in Physics.

Whitman College
2020

LIGO-P2000185

Certificate of Approval

This is to certify that the accompanying thesis by Thomas Robert Brooks Harris has been accepted in partial fulfillment of the requirements for graduation with Honors in Physics.

_____
Prof. Frederick Moore

Whitman College
May 20, 2020

Table of Contents

Acknowledgements

I am very grateful to my previous research mentor, Dr. Gregory Vaughn-Ogin, for enabling me to work with LIGO's Detector Characterization group in the summer of 2018 and get my start in physics research. I am also very grateful to my present mentor, Dr. Gregory Mendell of the LIGO Hanford Observatory, who has challenged and supported me every step of the way over these last two years. I would also like to sincerely thank my adviser, Prof. Frederick Moore, for his guidance throughout my college career and for making it possible for me to work on continued LIGO research during my senior year.

Many thanks to Whitman College for funding a summer of undergraduate research in 2018, which enabled me to make connections at LIGO and start on a path towards this thesis project. LIGO is funded by the U.S. National Science Foundation.

Abstract

The Laser Interferometer Gravitational-Wave Observatory (LIGO) detectors are designed to capture and record gravitational wave strain signals from astrophysical sources. The Fscan family of algorithms is a key signal processing tool for analyzing the spectral content of LIGO data. An area of ongoing investigation are features of the LIGO Hanford strain data in the sub-100Hz range that are not fully understood. This work utilizes a statistically-oriented, data-mining approach to analyze three related but distinct subjects in LIGO Hanford data from the O3-era observing run: the coherence of auxiliary data channels with the strain, the potential for coherence between the Hanford and Livingston interferometers not caused by astrophysical sources, and the shifting density of high-power spectral lines in strain data.

List of Figures

Chapter 1

Introduction and Background

Gravitational Waves

Gravitational waves (GWs) are ripples in space-time caused by acceleration of sources of gravity. When an object has a time-changing quadrupole moment, it radiates gravitational waves [1]. Gravitational waves are transverse waves and are described in terms of plus (+) and cross (×) polarizations.

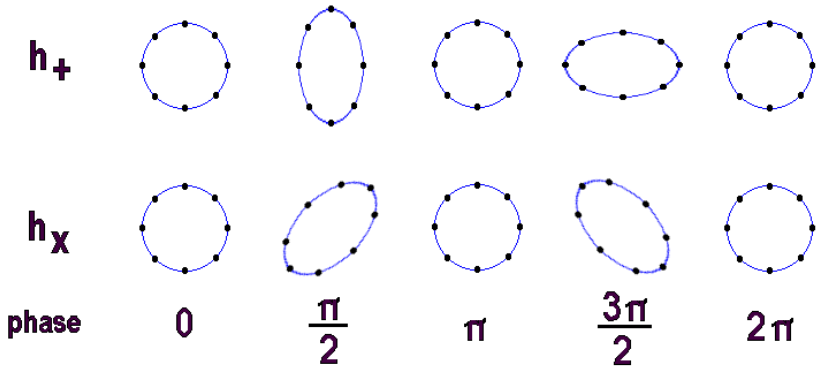

Figure 1: Gravitational wave + polarization and × polarization, depicted for GWs entering the page: the distances between the points change as shown as a function of GW phase [2]

The amplitude of a gravitational wave is usually characterized by the wave's *strain*. As a function of time, conventionally the strain is called *h(t)*. The strain is defined as the fractional change in distance between test masses as the GW passes through the masses and the space between them [3],

$$\frac{\delta L(t)}{L} = F_+ h_+(t) + F_\times h_\times(t) \equiv h(t) \qquad \text{(Eqn. 1)}$$

where δL denotes the amount by which the length between the test masses changes, L denotes the original distance between the test masses, F_+ and $F_\times$ are the antenna patterns of the detectors, and h_+ and $h_\times$ are the + polarization and × polarization. For reference, the measured strain amplitude of the first GW detection event (GW150914) was estimated to vary between roughly 0.5×10^{-21} and 1×10^{-21} during the inspiral in the fifth of a second immediately prior to the merger of two black holes [4]. This strain amplitude is on the higher end of what is expected from most compact binary coalescence GW transient signals. For more information on the general relativity of GWs, see [1] and [3].

As of the time of this writing, one formal GW catalog (GWTC-1) has been published (see Fig. 2) [5]. This catalog describes 11 confident GW detection events. Of those 11 GWs, 10 were determined to have originated from binary black hole inspirals and mergers, and 1 was determined to have originated from a binary neutron star inspiral and merger. The results in GWTC-1 were obtained by matched filtered searches using hundreds of thousands of GR-based waveform templates and by an unmodeled search for transient signals without a specific waveform model. Candidate events with significant signal-to-noise ratios (SNR) compared to background and probability of astrophysical origin greater than 50% were cataloged as GW events. The time spent in the LIGO band and the maximum GW frequency reached are used to determine the component masses of the system, while the SNR indicates the distance from Earth to the source. All the binary black hole events had duration of less than 2 seconds. The binary neutron star event spent 100 seconds in the LIGO band and was also observed electromagnetically. The total mass of the black holes systems in GWTC-1 ranged from 18 to 85 solar masses and the distances were from 300 to 3000 Mpc. The binary neutron star was also observed as a

weak gamma-ray burst and a kilonova in a galaxy 40 Mpc away, making it one of the most spectacular multi-messenger astrophysical discoveries to date. Since the publication of GWTC-1, a Gravitational-Wave Candidate Event Database website called GraceDB has publicly announced high-probability GW candidate events [6].

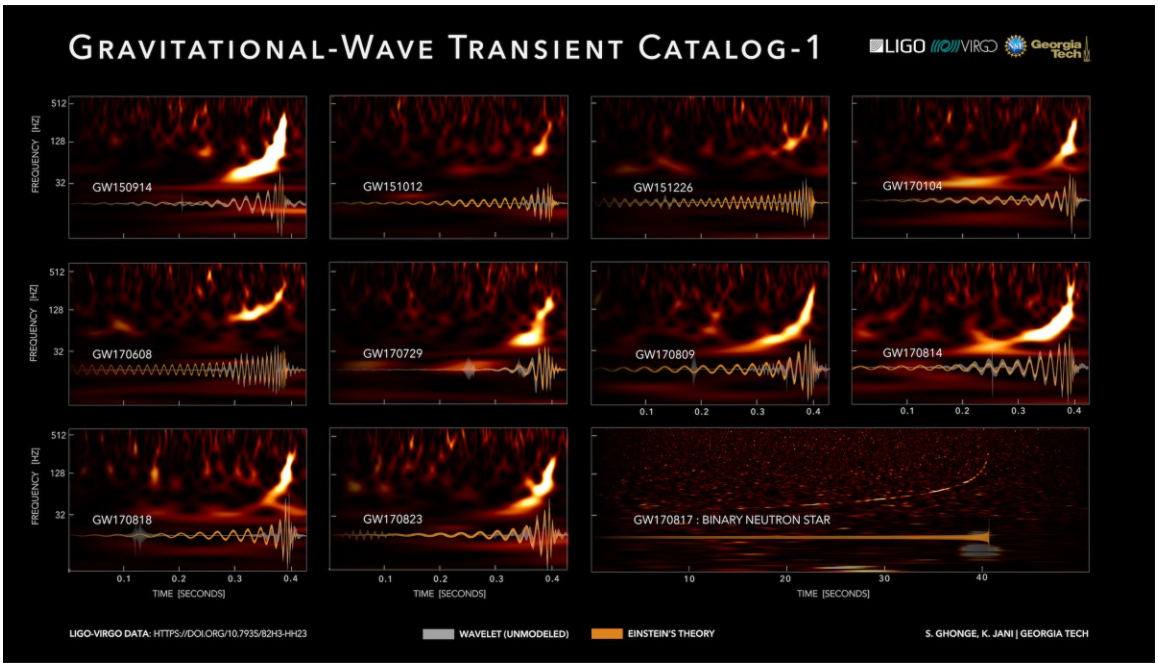

Figure 2: A poster representation of the 11 GW events discussed in GWTC-1. The GWs are shown here as spectrograms, with frequency on the y-axis and time on the x-axis [7].

The LIGO Detectors

The LIGO GW-detecting observatories are located in Hanford, WA and Livingston, LA. The observatories are modified Michelson interferometers with Fabry-Pérot cavities in the 4-kilometer arms, as seen in Fig. 3 [4] [8]. The use of multiple observatories to enable coincident detection of waveforms at widely separated locations (including collaboration with the Italian Virgo Collaboration) is key to making confident GW detections and to triangulating the origins of GW signals. The observatories ended

the O3 observing run in March 2020; O3 began in April 2019 and was split into two parts, O3a and O3b.

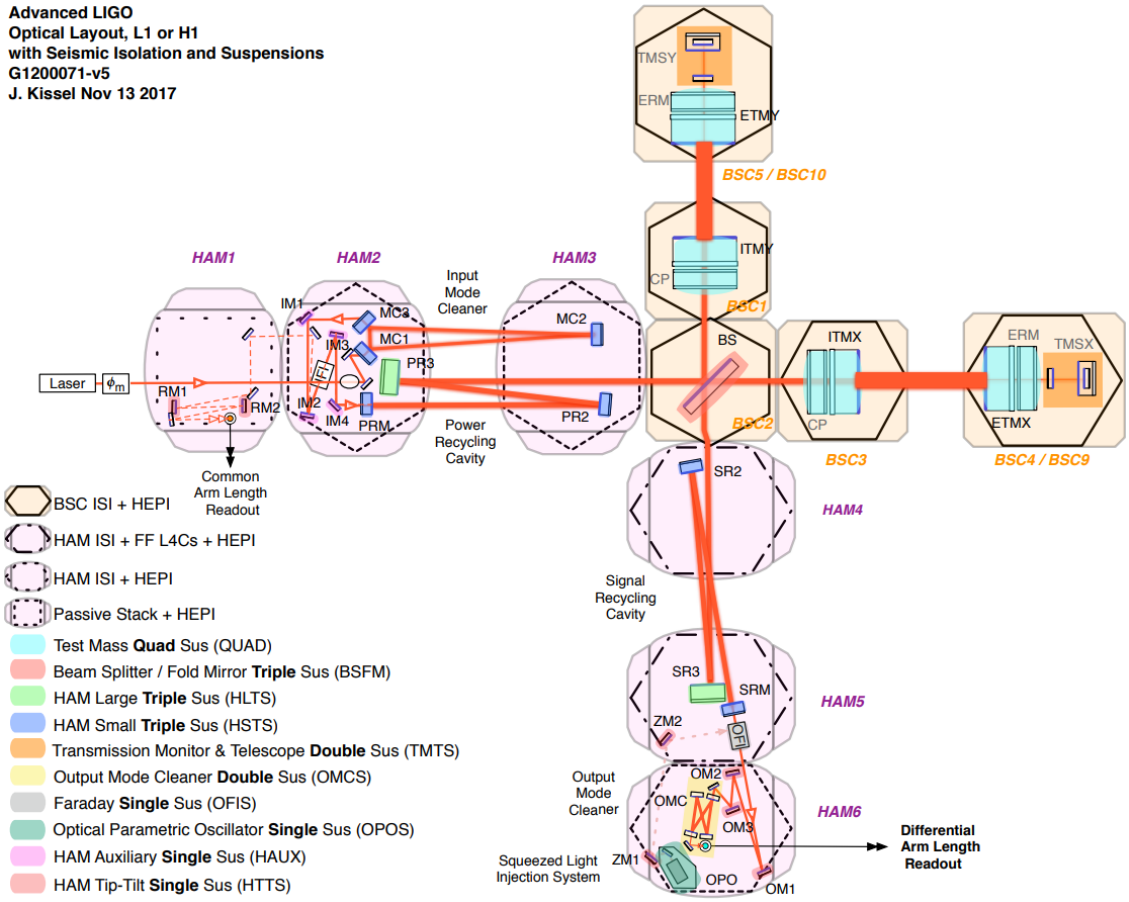

Figure 3: LIGO Hanford Optical Layout [9]

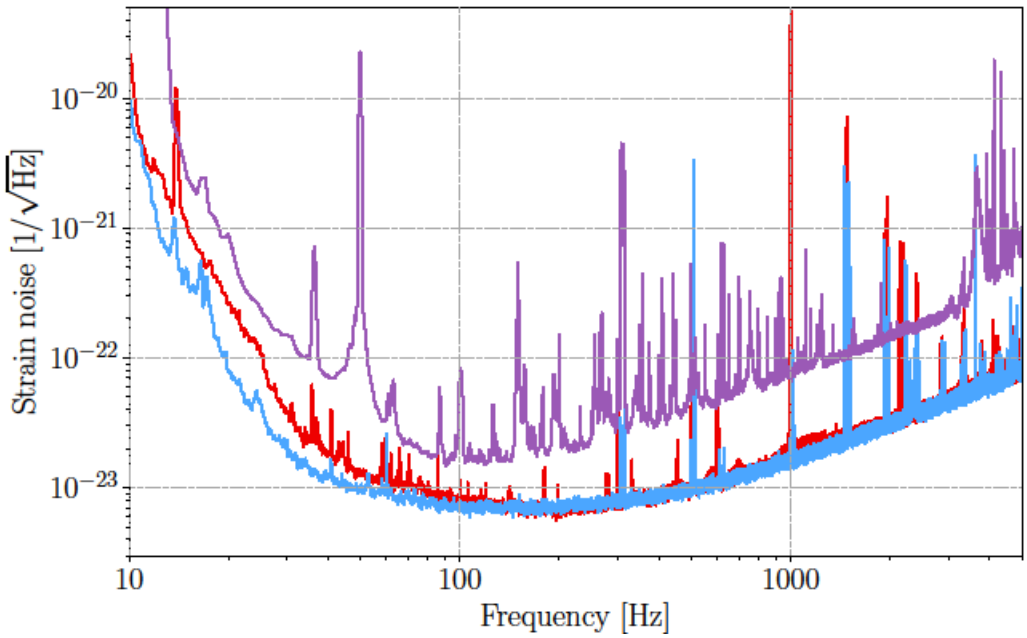

Figure 4: Amplitude spectral density of the strain sensitivity of the Virgo (purple), LIGO Hanford (red), and LIGO Livingston (blue) detectors. The curves are representative of the best performance of each detector during observing run O2. [5]

The LIGO detectors have an optimal strain sensitivity on the order of 10^{-23} $1/\sqrt{\text{Hz}}$ (see the definition of spectral density in the next section for information about units), which translates to a length change on the order of 10^{-20} m in the detector's most sensitive frequency band due to the 4-kilometer arms (see Fig. 4). Many isolation systems and noise-reducing design choices are necessary to maintain the detector sensitivity required to confidently capture data on the small strain amplitudes of gravitational waves. In addition to instrumental noise, the physical environment of the detector is a substantial noise source. The locations of some of the Physical Environment Monitoring (PEM) sensors is included here in Fig. 5 to illustrate the number and variety of witnesses there are for potential noise sources. The data recorded by sensors like these is recorded in interferometer "auxiliary channels." Monitoring of thousands of auxiliary channels is

used to characterize the state of the detector and screen for terrestrial disturbances. Analysis of the strain channel and auxiliary channels determines the quality of the data and provides vetoes on poor quality times or frequency bands.

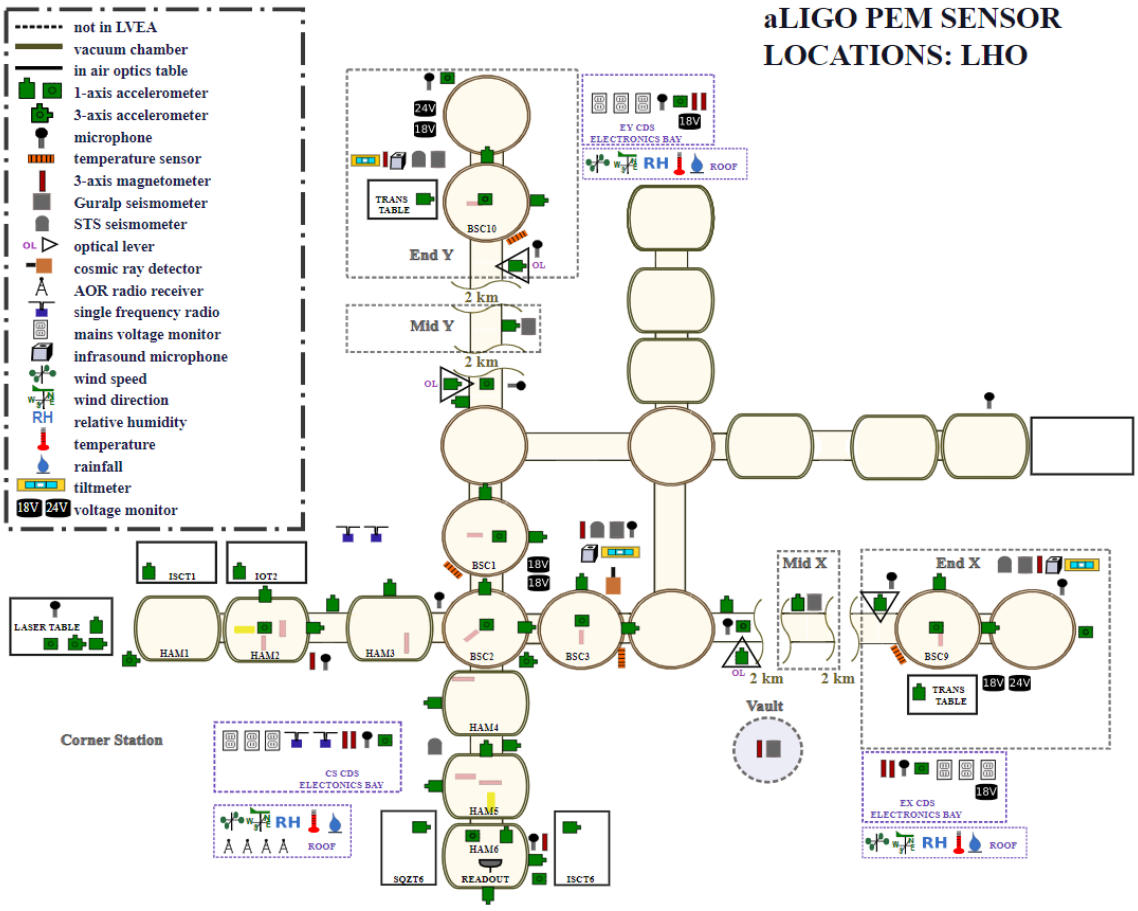

Figure 5: LIGO Hanford Physical Environment Monitoring (PEM) sensor layout [10]

Signal Processing and Fscan

Spectral analysis is central to much of LIGO's research. A primary tool used for detector characterization efforts is the Fscan family of algorithms [11]. Fscans are primarily used to find spectral lines in the strain channel and auxiliary channels and to calculate coherences between the strain channel and auxiliary channels. An automated system in the LIGO servers runs the Fscans on a daily, weekly, and monthly basis. The

Fscan driver script creates and organizes sets of Short-Time Fourier transforms (SFTs). The SFTs are so named because they are Fast Fourier Transforms (FFTs) over shorter chunks of time (typically 1800 s in duration) that get stitched together as the building blocks for very long duration FFTs by LIGO's Continuous Wave working group, among other things.

FFTs are algorithmic implementations one of the most fundamental signal processing and spectral analysis tools, the Discrete Fourier Transform (DFT), which is defined in LIGO as

$$\tilde{x}_k = \sum_{j=0}^{N-1} x_j e^{-i2\pi jk/N} \qquad \text{(Eqn. 2)}$$

where k is a frequency domain index, j is a time domain index, x_j is a data point in the time domain, N is the total number of data points, and i is the imaginary number $\sqrt{-1}$ [12]. For data sampled over time T, the frequency resolution $\Delta f = 1/T$ is the width between neighboring frequency "bins" in the frequency domain. The Fscans have a frequency resolution of 1/(1800 s) = (1/1800) Hz. The frequency associated with any given index k is $f_k = k / T$. The highest possible frequency about which one can have meaningful information from a DFT is the Nyquist frequency, which is defined as half of the sampling frequency ($f_{Nyquist} = N / (2T)$).

Two problems associated with the information provided by the DFT are aliasing and spectral leakage. Information in the useful band below the Nyquist frequency is aliased into the frequencies above the Nyquist frequency, and disturbances above the Nyquist frequency are also aliased into this useful band unless filtering is applied before digitizing the data. Spectral leakage is caused by the finite amount of time used to

generate DFTs. Spectral disturbances at frequencies that do not have an integer number of cycles during the DFT sample time leak power into neighboring frequency bins. To combat aliasing and spectral leakage, Hann windowing is applied to Fscan data [12]. For more information on spectral leakage, see [13].

An important feature of the DFT is that DFT sinusoids are orthogonal:

$$\sum_{j=0}^{N-1} e^{-i2\pi jk'/N} e^{-i2\pi jk/N} = N\delta_{kk'} \qquad \text{(Eqn. 3)}$$

This can be shown by application of a geometric series to the left side of Eqn. 3 [12].

For two signals *x* and *y*, the correlation (or cross-correlation) of *x* and *y* is defined as:

$$c_{j'} = \sum_{j=0}^{N-1} x_j y_{j+j'} \qquad \text{(Eqn. 4)}$$

where *j* and *j'* are time-domain indices, as before [12]. In the frequency domain, correlation is defined as:

$$\tilde{c}_k = \tilde{x}_k \tilde{y}_k^* \qquad \text{(Eqn. 5)}$$

where *k* indicates the frequency domain index, as before. If $x = y$, then this is called the autocorrelation of *x*. Signal correlation is foundational to several key signal processing tools used by the Fscans, namely signal coherence and power spectral densities.

The absolute square of a Fourier transform is termed "power" in signal processing. The power spectral density (PSD) estimation P_k utilizes the absolute square of DFTs to describe the power present in *k*-indexed frequency bins:

$$P_k = \frac{2\langle|\tilde{x}_k|^2\rangle \Delta t^2}{T} \qquad \text{(Eqn. 6)}$$

where Δt is the time between samples in the time j-indexed domain, T is the total sampling time, and angled brackets denote averaging [12]. This is a "one-sided" PSD that is defined for positive frequencies; the factor of 2 accounts for power from negative frequencies. Without averaging, Eqn. 6 is called a periodogram, so the PSD is estimated by averaging periodograms. A comparison of signal PSD or power versus frequency is called that signal's spectrum. The square root of the PSD is called the amplitude spectral density (ASD). The ASD is used to give units that can be compared more easily with time domain data [14]; the ASD is generally how detector strain sensitivity is reported, as in Fig. 4 previously.

The Fscans display spectra as graphs of power ($|\tilde{x}_k|^2$) versus frequency, normalized with a running median to make the average power equal to one (see Fig. 6). Recall that the frequency resolution of the Fscans is (1/1800) Hz. The Fscans also generate spectrograms, which are time-frequency plots of the power spectral density (see Fig. 7).

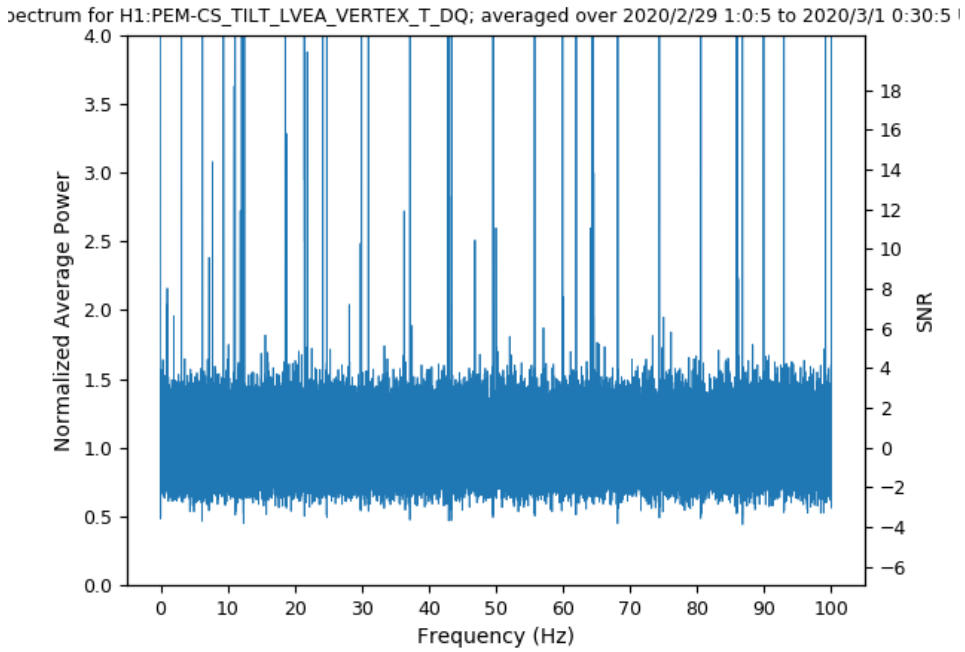

Figure 6: Illustrative sample of a normalized Fscan spectrum for one day of tiltmeter channel data [15]

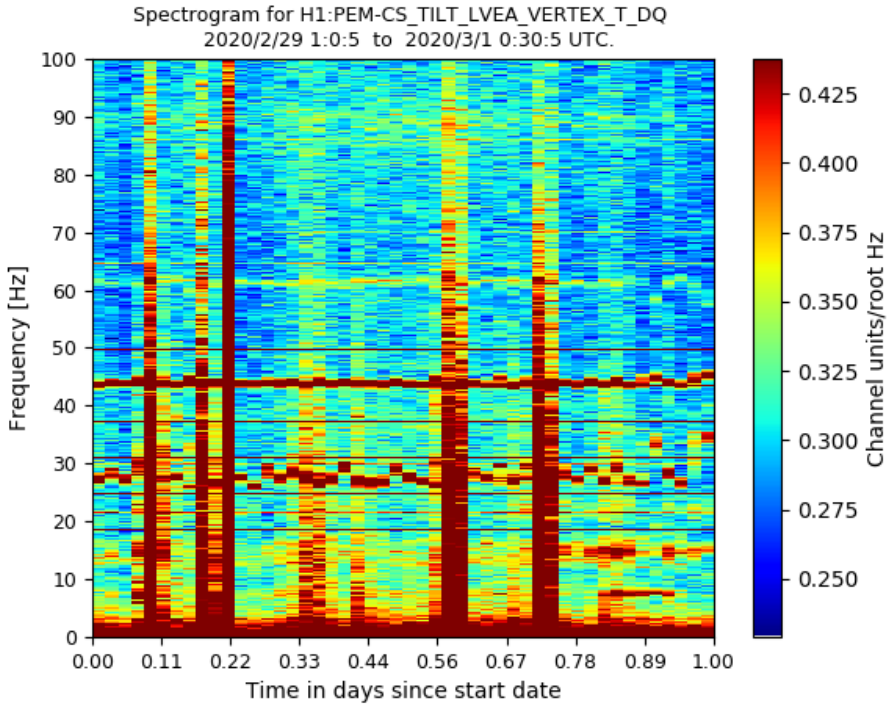

Figure 7: Illustrative sample of an Fscan spectrogram for one day of tiltmeter channel data [15]

The coherence (short for magnitude-squared coherence) $\hat{c}$ between two channels A and B is defined as

$$\hat{c} = \frac{\left|\langle \tilde{A}\tilde{B}^{*} \rangle\right|^{2}}{\langle \left|\tilde{A}\right|^{2} \rangle \langle \left|\tilde{B}\right|^{2} \rangle} \qquad \text{(Eqn. 7)}$$

where $\tilde{A} = a\, e^{i\Phi_1}$ and $\tilde{B} = b\, e^{i\Phi_2}$ and angled brackets denote averaging [12]. In terms of signal correlation, the coherence is the squared magnitude of the average of the correlation between A and B (numerator) divided by the product of the average autocorrelations of A and B (denominator). The coherence can also be expressed in the form

$$\hat{c} = \frac{|\langle\, ab\, [\, cos(\Phi_1 - \Phi_2) + i\, sin(\Phi_1 - \Phi_2)]\, \rangle|^{2}}{\langle a^{2} \rangle \langle b^{2} \rangle} \qquad \text{(Eqn. 8)}$$

As the equation above makes apparent, a constant phase difference will cause the coherence to tend towards 1, whereas a random relationship between the phases will cause the magnitude of the average to tend towards 0. In this way, the coherence between two signals is a measure of the constancy of the phase difference between them. Two important properties of the Fscans that minimize the impact of outliers in the product *ab* (which can produce anomalous coherences) are the thousands of averages done on data spanning an entire month, and the high frequency resolution of the individual SFTs.

Chapter 2

Data Mining Projects

This discussion now turns to three projects designed to mine the output data of the existing Fscan infrastructure for useful information that can be presented in a more human-digestible format, along with leveraging the Fscans in a new way to do signal coherence analysis across interferometers.

High Coherence Dictionary

One area of investigation is examining coherences between auxiliary channel data and the gravitational wave strain channel. For context, some auxiliary channels are considered by Detector Characterization and related LIGO groups to be "safe," and some are considered "unsafe" [16]. Safe channels are channels in which one would not expect an astrophysical signal in the strain to couple into the data recorded by that channel. Unsafe channels are channels in which a signal in the strain channel could couple into the channel's data, in addition to *h(t)* [11]. Whether or not an auxiliary channel couples with the strain channel to a sufficient degree to be considered unsafe is determined by various metrics, one of which is examination of data in auxiliary channels following "injections" of waveforms into the strain channel. If a waveform injected into the strain channel by an intentional manipulation of the test mass mirrors is found to be highly coherent with an auxiliary channel, then the safety of that auxiliary channel will come into question. One example of unsafe auxiliary channels are those that are coupled to the actuation on the detector mirrors, such as certain magnetometers near the electromagnets used to control the length of the interferometer arms.

A difficulty with using information from the Fscans to help determine channel safety or find unexpected coupling between the strain channel and auxiliary channels is that there is a large daily volume of coherence data produced by the Fscan system. The Fscans produce thousands of daily plots of the coherence between the strain channel and a subset of auxiliary channels. Historically, there has been too much information produced about coherences on a daily, weekly, and monthly basis for a person to be able to sit down and identify by eye what channels have unusually high coherence with the strain channel – the volume is too great. Besides the problem of volume, determining what constitutes an "unusually" high coherence is not a task that a person can consistently perform by eye. Determining the threshold for a coherence to be "unusually" high should be a flexible metric that behaves in a statistically consistent manner.

An existing empirical study of the false alarm rate for coherences [17] found that for N averages used to calculate a magnitude-squared coherence value, the false alarm rate varied as described in Eqn. 9:

$$\text{Probability}(X_{obs} > X) = P = (1 - X)^{k(N-1)} \qquad \text{(Eqn. 9)}$$

where X_{obs} is the observed coherence value, X is any possible coherence value (note that coherence values range from 0 to 1), and k is a factor very close to 1 that varies slowly with N when data is Hann windowed. If we choose a target false alarm rate, we can solve this equation for a coherence value X and use that as our threshold for coherences between strain and auxiliary channels that are statistically unlikely to have happened by random chance. Setting k = 1 (because k is close to 1, about 0.95 for Hann windowing with 50% overlap) and rearranging Eqn. 9, we find that

$$X = 1 - P^{\frac{1}{(N-1)}} \qquad \text{(Eqn. 10)}$$

is a statistically consistent metric for a threshold for unusually high coherences. Upon choosing a cutoff false alarm rate probability and finding the number of averages used to make a particular set of coherence output for the strain channel and some auxiliary channel, it is possible to determine which coherences are higher than one would expect to find by chance based on the chosen probability. Typically, this probability cutoff is set to be on the order of 10^{-10}, about five orders of magnitude smaller than 1 / (the number of frequency bins in the 0-100Hz range of the Fscans) = 1 / 180,000.

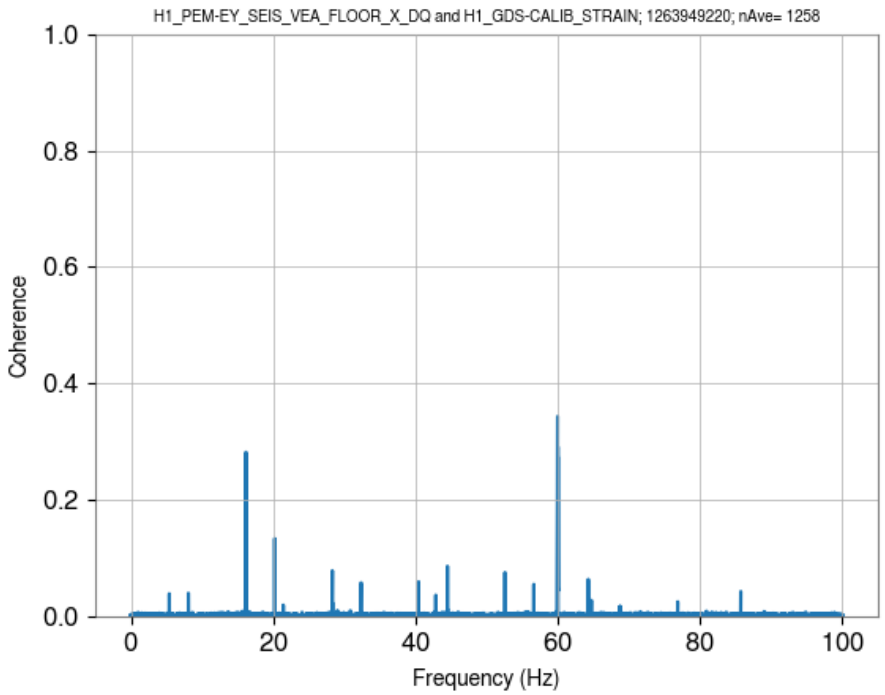

Figure 8: Typical Hanford seismometer channel and strain month-long coherence plot example [15]

For illustration, see Fig. 8, which shows the month-long coherence between a seismometer and the strain channel after 1258 averages. By Eqn. 10, a coherence above ~1.9×10^{-2} is highly unlikely by random chance. On the other hand, mechanical vibrations tend to damp out and are unlikely coherent for a month. Thus, the significant coherences seen in Fig. 8 are likely unwanted digital effects produced in computers. Efforts to track

down where this occurs in the electronics are aided by knowing which channels suffer from this problem at a given frequency.

I created a series of Python algorithms that can take existing Fscan coherence data and apply the method described above to find unusually high coherences. The code for this project, `fscan_coherence_analyzer.py`, is in Appendix A to this paper. This script is particularly useful because it can present the results in multiple ways. If one is only interested in viewing the high coherences for individual channels over a certain time frame, it's possible to look at only that limited subset of data. Using command line arguments, it is also possible to format the output of the script as a frequency "dictionary." Producing this dictionary is the primary utility of `fscan_coherence_analyzer.py`. Bands of frequencies are listed in the dictionary output file, and any auxiliary channels with high coherence to the strain channel in that frequency band are listed, along with their coherence values. The goal of formatting the output in this way is to allow persons working on detector characterization to tackle the problem of mystery noise sources in the Hanford interferometer's sub-100Hz band. For example, the coherence dictionary output file sample in Appendix E (which uses data from February 2020; it is labelled as March 1, 2020 because it uses data from the month prior) lists a number of auxiliary channels that are coherent with the strain channel in a narrow band less than 2mHz wide around 11.394Hz. Resources used to track mystery spectral lines and sets of spectral lines called "combs" in the strain data list a potential comb at 11.39495Hz, suggesting a possible match [18]. The fact that the match occurs in channels in the x-arm of the detector is an important clue from the dictionary as to which computers might be responsible for this comb of lines.

For the sake of convenience, certain sets of information can be omitted at the discretion of the person running the script. If they are not interested in seeing integer frequencies in the output, or not interested in seeing a band around 60Hz due to high coherences with power supplies, it is possible to suppress those outputs.

A potential future application of this coherence analyzer could be generating comparisons of the number of unusually high coherences in analogous channels at LHO and LLO. This could be used to quantify exactly how much worse the coupling between strain and certain types of noise sources is at one site compared to the other.

Hanford-Livingston Coherence Investigations

The Fscan architecture is capable of computing long-term coherences using many SFTs – that's how it generates monthly summary plots. However, the application of this has historically only been to target the coherence between the strain channel at either interferometer and an auxiliary channel at that interferometer. It was not used to generate inter-site coherence analyses. Because of curiosity surrounding the potential application of the Fscans to look directly at coherences between the interferometers, I worked on `script_modmultidayfscan.py` (Appendix B) and `segment_intersector.py` (Appendix C) to examine coherences between Hanford and Livingston GW strain channels. A main goal of this project was investigating if the strain channels of the interferometers are significantly coherent over long periods of time, on the order of a month. Because of the physical separation of the interferometers, there shouldn't be a mechanical oscillations that manage to remain coherent between the two sites by chance on the scale of a month.

Generating coherences between the interferometers using Fscans requires several highly modified versions of other Fscan scripts in order to work, most of which are not included in this work or its appendices (but are saved on the LIGO Hanford servers at [19]). This is a new type of Fscan data product. The role of `segment_intersector.py` is the most explicit. At different times, data from the interferometers is flagged as belonging in different categories – for example, marking whether or not the interferometer is on, if the interferometer is in low-noise mode, if the data being produced is ready for use in analysis and searches, etc. This script finds the intersection between the times when Hanford and Livingston are producing analysis-ready strain data and acts as a substitute for the normal process used to find segments in the Fscans. Normally, the Fscan algorithms only need to worry about the data form one interferometer, so adding this script was necessary to utilize data only from segments of time when both interferometers were locked and producing analysis-ready data. Failing to use this script could result in good data from one interferometer overlapping with terrible or non-existent data from the other interferometer, which could significantly throw off the results of the Fscans.

Over a month of coherence data from the intersection of Hanford and Livingston analysis-ready times was produced, and a summary month Fscan was generated on this data to obtain month-long Hanford/Livingston coherence information [20]. Some of the daily Hanford/Livingston coherence plots had unusual characteristics and bizarrely high coherences, visible in the following plots [21]. The reason for these unusual features is not fully understood, but we have suspicions that one or two very loud, very bad SFTs on certain days had an excessive effect on the coherence values for the whole day.

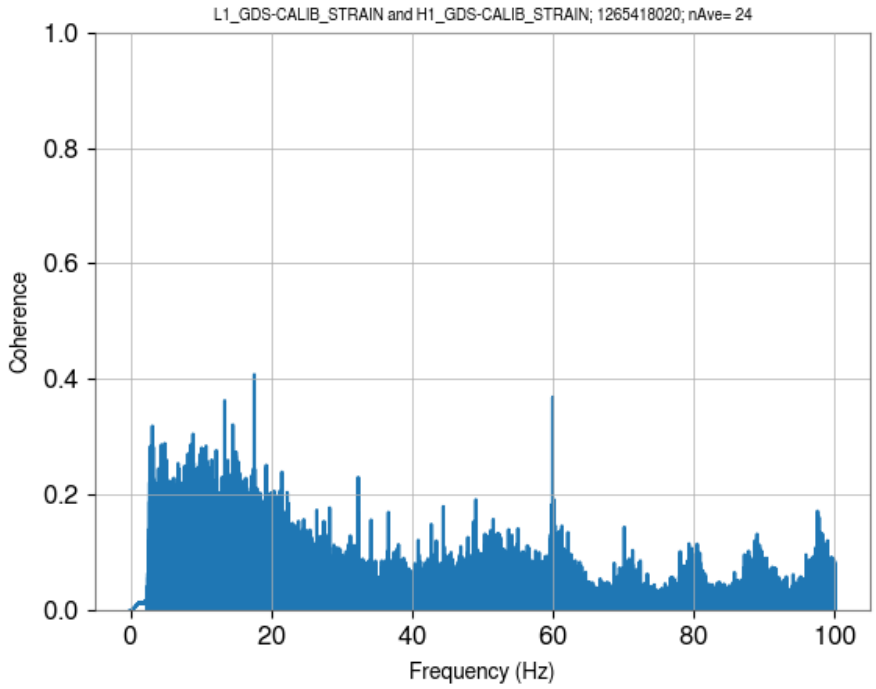

Figure 9: Coherence between Hanford and Livingston on February 11, 2020 [19]

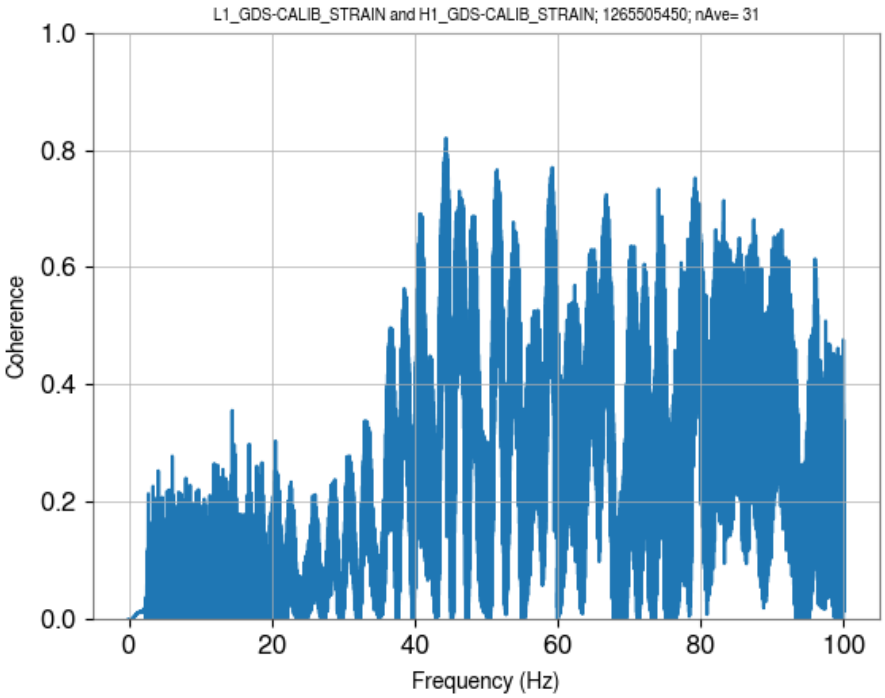

Figure 10: Coherence between Hanford and Livingston on February 12, 2020 [19]

The month-long Hanford/Livingston coherence shown in Fig. 11 was closer to what was expected for these plots. Many of the coherence values were low, nearly plotting a straight line across the bottom of the coherence plot. There are some appreciable bumps in the coherence, however, so all of the data points in this plot were subjected to the significant coherence analysis described previously in the coherence dictionary section.

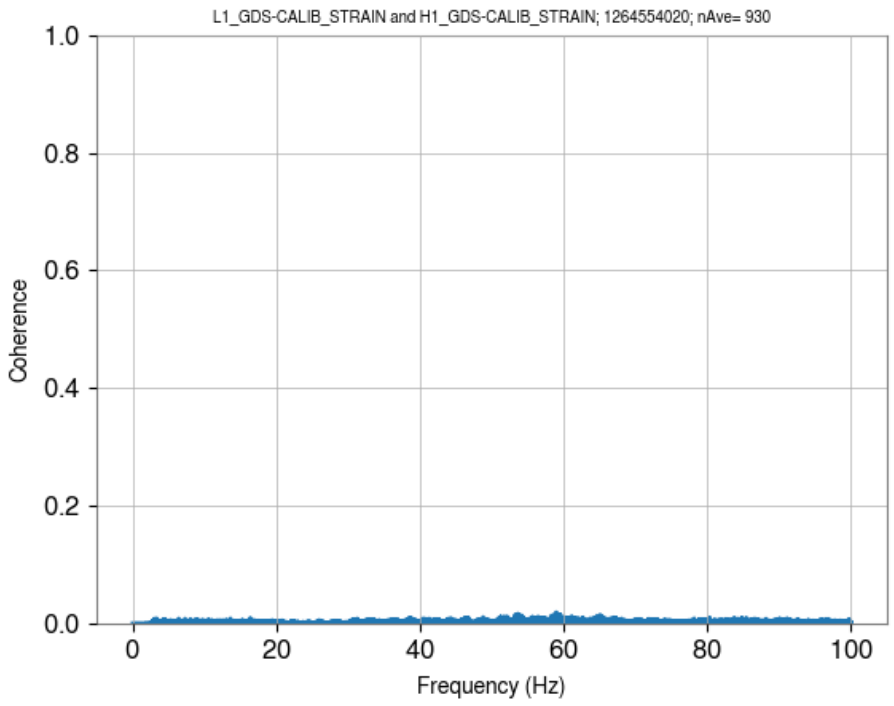

Figure 11: Coherence between Hanford and Livingston for all of February 2020 [20]

Applying Eqn. 10 to the month-long Hanford/Livingston coherence values, where N = 930 averages and cutoff probability was on the order of 5×10^{-7}, resulted in a threshold coherence value of ~0.0154. There are about 20 significant frequencies with coherences greater than this threshold in the month-long Hanford/Livingston coherence. All these significant frequencies are between 53.3Hz and 65.1Hz. The reason for 20 frequencies

remaining significantly coherent over an entire month is unclear. They are all near 60Hz, which suggests the possibility that they could be sidebands of 60Hz and electrical in origin. It is possible that some part of the data being generated by the Fscan SFTs is introducing coherence that should not be there. For now, the reason for these coherences being significant at all is an unresolved question.

High-Power Line Density Tracker

One of the interesting features of the LIGO Hanford strain data is that it is consistently noisier than the LIGO Livingston data in the sub-100Hz range. Comparing spectra by eye, it is easy to tell for most of O3 that the LIGO Hanford strain spectrum had more high-power lines than the LIGO Livingston strain data (pictured on the next page) [15]. This “forest of lines” in the sub-100Hz LIGO Hanford data causes many strain spectra plots to appear messy. For context, it also does not appear that there was a distinct day/week/month in O3 (or before) that marked the onset of the Hanford forest of lines [22]. It is difficult to identify which days have a worse proliferation of high-power spectral lines than the average day at Hanford – by eye, some days look better or worse than others, but that is not a systematic and objective evaluation. Most problematically, if every day has a “bad-looking” spectral plot, human eyes will miss more subtle structural changes in the forest of lines. See Figs. 12 and 13 for a visual comparison of month-long Hanford and Livingston strain spectra.

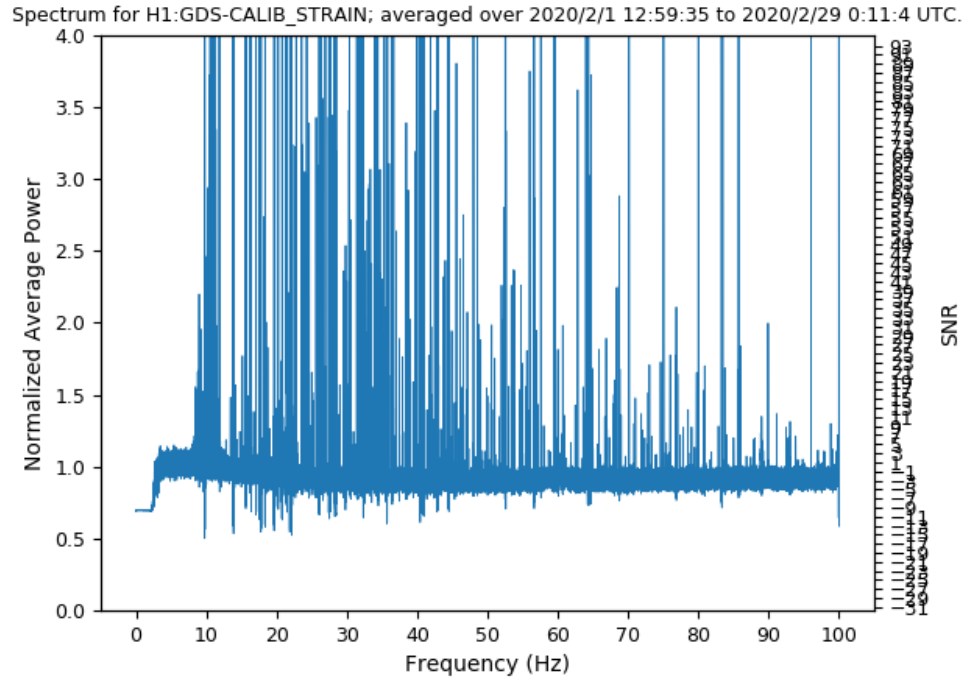

Figure 12: LIGO Hanford Fscan Strain Spectrum, Feb. 2020 [20]

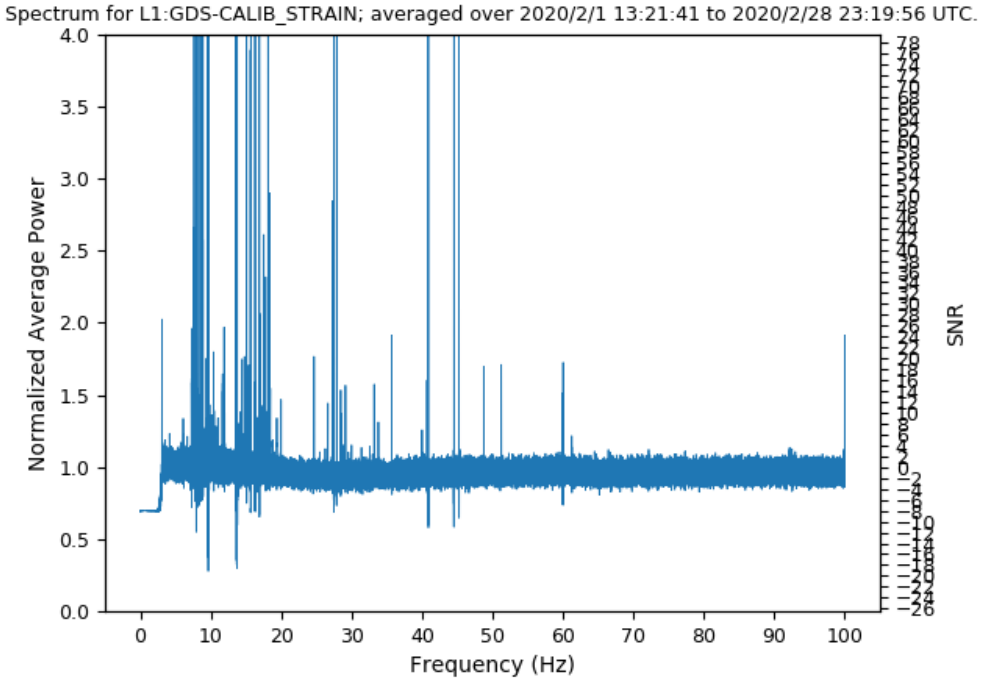

Figure 13: LIGO Livingston Fscan Strain Spectrum, Feb. 2020 [20]

To address this, I developed an algorithm (`fscan_lines.py`) that parses month-long batches of daily Hanford spectra data and presents useful information about line density [21]. The purpose of this tool is to address the lack of a human-legible summary of Hanford's changing line density. If one can generate criteria for what constitutes a "high-power" line, one can count the number of high-power lines in a spectral plot. Line density is a shorthand way of referring to the number of high-power lines in a spectrum (i.e. number of lines per day, per week, per month, etc.).

To determine the threshold criteria for high-power lines in a given strain spectrum, one can utilize χ^2 statistical methods [13] [12]. Consider noise $\tilde{n}$ in the strain as a complex signal for a particular frequency bin, and for the moment also consider the case of just one SFT contributing to the Fscan data. For Gaussian noise signals, the distributions of the real (x) and imaginary (y) parts of $\tilde{n}$ are each individually Gaussian. For the FFT of noise $\tilde{n}$ expressed as

$$\tilde{\mathrm{n}} = x + iy \qquad \text{(Eqn. 11)}$$

if one treats only noise as contributing to the power ρ in each frequency bin in the spectrum of the strain, then it follows that

$$\rho = |\tilde{\mathrm{n}}|^2 = x^2 + y^2 \qquad \text{(Eqn. 12)}$$

with proper normalization of x and y. The power can be expressed as a sum of squares of Gaussian-distributed variables, which is a type of statistical situation where χ^2 statistical methods apply. This can be shown through a simple change of variables from the combined probability distribution of x and y:

$$P(x,y)dxdy = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dxdy \qquad \text{(Eqn. 13)}$$

Let $x = r\,cos(\theta)$, $y = r\,sin(\theta)$, $dxdy = rdrd\theta$ and integrate over $0 < \theta < \frac{\pi}{2}$ to find the Rayleigh distribution,

$$P(r)dr = re^{-r^2/2}dr \qquad \text{(Eqn. 14)}$$

which converts to a χ^2 distribution for ρ with 2 degrees of freedom with the substitutions $\rho = r^2$ and $d\rho = 2rdr$:

$$P(\rho)d\rho = \frac{1}{2}e^{-\rho/2}d\rho \qquad \text{(Eqn. 15)}$$

This derivation generalizes to the case where more than one SFT is contributing to a spectrum. With n-many SFTs, each SFT has a real and complex part contributing 2 degrees of freedom. Then the probability as a function of power takes on the form of the χ^2 distribution for ρ with 2n degrees of freedom:

$$P(\rho)d\rho = \frac{1}{2^{(\frac{n}{2})}\,\Gamma(n/2)}\rho^{(\frac{n}{2}-1)}e^{-\rho/2}d\rho \qquad \text{(Eqn. 16)}$$

This line of reasoning is employed in `fscan_lines.py` to decide what the constitutes a high-power spectral line. A probability is supplied to the script as a command line argument (say a 1 in a million chance, or 1×10^{-6}). This is the probability of a line being above the power threshold. A chosen month is also supplied to the script by the command line. The script examines each daily Fscan output for all days in the chosen month for which data is available. For each available day, the script finds how many SFTs were used in the Fscan strain spectra processing from that day. It then calculates the inverse survival function of the χ^2 distribution using the supplied probability and degrees of freedom equal to twice the number of SFTs. Because Fscan spectra power values are normalized to a median value of 1, the power supplied by the inverse survival function is normalized and then used as a threshold for deciding if a frequency bin has a significantly

high power associated with it. For example, in Appendix G, the sample output for February 2020 data shows that on February 28 the normalized threshold power was ~2.05 and there were 1012 frequency bins with powers higher than the threshold.

The `fscan_lines.py` script computes the average number of daily high-power lines for the month, and then the standard deviation of the number of daily high-power lines. The script lists which days in the month had the worst outliers (two standard deviations or more above the average daily high-power line density), providing an objective metric for evaluating what days the forest of lines is appreciably worse than usual. Based on several months of O3 data, the average daily line density is on the order of hundreds, typically ranging between 500-2000 high-power lines in the Hanford strain spectrum per day.

Now that the infrastructure for this tool is in place, there are a number of ways in which it could be utilized for future Fscan data mining. For the time being, it is an easily accessible and human-legible snapshot into the varying messiness of noise in the strain data. But in the future, this code could easily be adapted to examine particular trends, such as which days (if any) of the typical Sunday-Saturday week tend to have worse forests of lines, for example.

One particularly exciting potential avenue for future investigation of lines could be the application of a machine learning algorithm to the daily plots of strain spectra. Machine learning applications to detector characterization activities are active areas of investigation [23]. In particular, the GravitySpy citizen science project has seen tremendous success in image-based training of "deep learning" algorithms with convolutional neural network layers to classify different morphologies of glitches in the

LIGO data, using citizen scientist assistance in developing training sets of images [24]. An application of image analysis-based machine learning algorithms to the strain spectra produced by Fscan would certainly be more modest in scope than the GravitySpy project, but nonetheless could potentially yield insight into more subtle variations in the forest of lines at Hanford over time.

Appendix A: `fscan_coherence_analyzer.py`

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
@author: Thomas Harris <harristr@whitman.edu>
"""

__author__ = 'Thomas Harris <harristr@whitman.edu>'

#Importing useful tools

import argparse
import sys
import os
import glob
import subprocess
from datetime import datetime

##############
# Functions #
##############

"""
Ideas for future:
- Integrate frequency bands greater than 100 Hz into script analysis.
- Cut off at exactly 7 significant figures in frequency list (instead of 6 after decimal)
"""

def get_channel_list(targetpath):
    """
    :param targetpath: The path to the folder containing channels for analysis
    :return: List containing strings of channel names
    """
    os.chdir(targetpath)  # Going to the target directory
    current_chan_path = subprocess.check_output("pwd").rstrip()
    print("Current working directory: " + current_chan_path)

    channel_list = subprocess.check_output("ls -
1 | cat", shell=True).rstrip().split("\n")

    for name in channel_list[:]:
        if name == "fscanChannels.html" or name == "H1_GDS-CALIB_STRAIN":
            channel_list.remove(name)

    print("Number of channels: " + str(len(channel_list)))

    return channel_list

def find_fscan_date(chan_path):
    """
    :param chan_path: The path to the channel being analyzed
    :return: date_string: Returns the string of the channel date.
    """
    date_found = False
    counter1_found = False
    date_syntax_counter = 1
```

```
    while not date_found:
        slash_check = chan_path[-date_syntax_counter]
        if slash_check == "/" and counter1_found == False:
            counter1 = date_syntax_counter
            counter1_found = True
        elif slash_check == "/":
            counter2 = date_syntax_counter
            date_found = True
        date_syntax_counter += 1

    date_string = chan_path[1 - counter2: -counter1]
    return date_string

def get_chan_name(file):
    """
    :param file: Name of the file containing high-coherence data for a channel
    :return: chan_name: name of the channel
    """
    chan_name = file.split("_and_")[0].split("_coherence_")[1]
    return chan_name

def write_dict_entry(outputfile, templist, include_60Hz_band, include_integer, channel_info_dict):
    """
    :param outputfile: file where dictionary output for bands is being written
    :param templist: list of information for a specific band
    :param include_60Hz_band: boolean for including 60Hz band or not
    :param include_integer: boolean for including integer frequencies or not
    :param channel_info_dict: {get_chan_name(coherencefilename): [signifcoherence, signifcohlist], ... }
    :return: None
    """
    if include_60Hz_band or not (templist[0] <= 60.0 <= templist[1]): #If shortened output, ignore band with 60 Hz
        if include_integer or int(templist[0]) == int(templist[1]):
            outputfile.write("\n# {0:6f} {1:6f} {2:6f}\n".format(templist[0], templist[1], templist[1] - templist[0]))
            for index in range(2, len(templist)):
                #Getting the max coherence value for given channel in given band
                coherence_list = []
                for triplet in channel_info_dict[templist[index]][1]:
                    if templist[0] <= triplet[1] <= templist[1]:
                        coherence_list.append(triplet[2])
                outputfile.write("    {0} {1}\n".format(templist[index], max(coherence_list)))

def analyze_channel(chanpath, outputpath, duration, significance_coefficient, include_integer, freq_cutoff):
    """
    :param chanpath: path to channel for analysis
    :param outputpath: where to write outputs
    :param duration: string (monthly_), (weekly_), or (daily_)
    :param significance_coefficient: positive float for use in significant coherence calculation
    :param include_integer: boolean for including integer frequencies or not
    :param freq_cutoff: float value of max difference between frequencies for them to be considered in the same band
    :return: {get_chan_name(coherencefilename): [signifcoherence, signifcohlist]}
    """
    os.chdir(chanpath)  # Going to the target directory
    current_chan_path = subprocess.check_output("pwd").rstrip()
```

```
108.    print("\nCurrent working directory: " + current_chan_path)
109.
110.    # Find number of SFTs used in coherence calculations:
111.    # First check that the coherence data exists in this frequency range, fail if not.
112.    prelim = subprocess.check_output(
113.        "cat logs/runCoherence*.out | grep 'Coherence Completed' | awk '{print $NF}'", she
    ll=True).rstrip()
114.    if prelim == None or prelim == "":
115.        print("\n## ## ## COHERENCE ANALYSIS FAILED FOR THIS CHANNEL. ## ## ##\n")
116.        return 1
117.    numSFTs = int(prelim)
118.    print("Number of SFTs: " + str(numSFTs))
119.
120.    # Find coherence data file
121.    coherencefilename = subprocess.check_output("ls -
    1 spec_0.00_100.00*coherence*.txt", shell=True).rstrip()
122.    print("Coherence file name: " + coherencefilename)
123.
124.    # Get number of coherence data points in file
125.    numcoherences = int(
126.        subprocess.check_output("cat -n " + coherencefilename + " | tail -
    1 | awk '{print $1}'", shell=True).rstrip())
127.    print("Number of coherences: " + str(numcoherences))
128.
129.    signifcoherence = 1.0 - ((1.0 / ((10 ** significance_coefficient) * float(numcoherence
    s))) ** (1.0 / float(numSFTs - 1)))
130.    print("Significant Coherence: " + str(signifcoherence))
131.
132.    # Acquiring list of highly coherent data from current channel
133.    signifcohlist = []
134.    chdata = open(coherencefilename, "r")
135.    line_index = 0
136.    for line in chdata:
137.        pair = line.split()
138.        pair = [float(value) for value in pair]
139.        if pair[1] >= signifcoherence:
140.            signifcohlist.append([line_index, pair[0], pair[1]]) #line index, frequency, c
    oherence
141.        line_index += 1
142.    chdata.close()
143.
144.    if not include_integer: # Abbreviating the output to have no integer frequencies if de
    sired
145.        for set in signifcohlist:
146.            if set[1].is_integer():
147.                signifcohlist.remove(set)
148.
149.
150.    print("Number of significant coherences: " + str(len(signifcohlist)))
151.    if len(signifcohlist) == 0:
152.        print("\nNo significant coherences for this channel.\n")
153.        return 0
154.    """
155.    for demo in range(len(signifcohlist)):
156.        if demo < 8 or demo > len(signifcohlist) - 9:
157.            print(signifcohlist[demo])
158.    """
159.
160.    # Finding bands of interest in highly coherent data
161.    bandsignifcohlist = []  # Initialize list of bands of coherent frequencies
162.    templist = []  # Initialize list for given band
```

```
    #print("\nCoherence band (templist) format: [freqmax] [cohmax] [freqlo] [freqhi]")

    for row in range(len(signifcohlist) - 1):
        if row == 0:  # First row
            currline = signifcohlist[row]
            nextline = signifcohlist[row + 1]
            templist.extend(currline[1:3] + [currline[1]] + [currline[1]])

        else:  # Any intermediate row
            currline = nextline
            nextline = signifcohlist[row + 1]

        if nextline[1] - currline[1] <= (freq_cutoff + 0.0001):  # If next line is within same frequency band
            if templist[1] < nextline[2]:  # Compare coherence of current line to max coherence
                templist[0] = nextline[1]
                templist[1] = nextline[2]
            templist[3] = nextline[1]
            if row == len(signifcohlist) - 2:  # Handle case where next row is last row
                #print("#### New BAND ####")
                #print(templist)
                bandsignifcohlist.append(templist)

        else:  # If not within same band
            #print("#### New BAND ####")
            #print(templist)
            templist[3] = currline[1]
            bandsignifcohlist.append(templist)
            templist = [nextline[1], nextline[2], nextline[1], nextline[1]]  # Prep templist for new band
            if row == len(signifcohlist) - 2:  # Handle case where next row is last row
                bandsignifcohlist.append(templist)

    #print(bandsignifcohlist)

    os.chdir(outputpath)  # Going to the output directory
    print("\nCurrent working directory: " + subprocess.check_output("pwd").rstrip())

    fscan_date = find_fscan_date(current_chan_path)

    #Writing the analysis output for this channel
    with open("signif_coherences_" + duration + fscan_date + "_" + coherencefilename, "w") as outputfile:
        outputfile.write("Significantly Coherent frequencies in " + coherencefilename + "\n")
        outputfile.write("Date of fscans: " + fscan_date + "\n")
        outputfile.write("Criteria for significant coherence: coherence >= " + str(signifcoherence) + "\n")
        outputfile.write("Number of significantly coherent frequencies: " + str(len(signifcohlist)) + "\n")
        if not include_integer:
            outputfile.write("This output is REDUCED to have no integer frequencies (argument include_integer == 'n')\n")
        else:
            outputfile.write("This output is FULL and includes integer frequencies (argument include_integer == 'y')\n")

        # Write a list of highly coherent frequency bands and their max coherences
        outputfile.write("\n## High coherence frequency bands ##\n"
```

```
                        "## Format: [most coherent frequency in band] [highest coherence in band] "
                        "[lowest band freq] [highest band freq] ##\n")
        for list in bandsignifcohlist:
            outputfile.write("{0:.6f} ".format(list[0]) +
                             "{0:.4f} ".format(list[1]) +
                             "{0:.6f} ".format(list[2]) +
                             "{0:.6f}\n".format(list[3]))

        # Write list of highly coherent frequencies and their coherences
        outputfile.write("\n## High coherence frequencies ##\n"
                        "## Format: [frequency] [coherence] ##\n")
        for frequency in signifcohlist:
            outputfile.write("{0:.6f} ".format(frequency[1]) +
                             "{0:.4f}\n".format(frequency[2]))

    outputfile.close()

    print("Highly coherent channel data recorded.")

    return {get_chan_name(coherencefilename): [signifcoherence, signifcohlist]}

def makedict(outputpath, duration, output_cutoff, freq_cutoff, band_cutoff, include_60Hz_band, include_integer, channel_info_dict):
    """
    :param outputpath: Where to write dictionary for set of channels
    :param duration: string (monthly_), (weekly_), or (daily_)
    :param output_cutoff: boolean for abbreviating data output or not
    :param freq_cutoff: float value of max difference between frequencies for them to be considered in the same band
    :param band_cutoff: max whole number of frequency bins between frequencies for them to be considered in the same band
    :param include_60Hz_band: whether or not to include a 60Hz band in the dictionary band summary
    :param include_integer: boolean for including integer frequencies or not
    :param channel_info_dict: {get_chan_name(coherencefilename): [signifcoherence, signifcohlist], ... }
    :return: None
    """
    print("\nMaking frequency dictionary in: " + outputpath)
    os.chdir(outputpath)  # Going to the output directory
    print("Current working directory: " + subprocess.check_output("pwd").rstrip())

    # Getting channel list
    channel_list = subprocess.check_output("ls -1 | cat | awk '{print $NF}'", shell=True).rstrip().split("\n")
    # Checking for non-empty list of output files
    if len(channel_list) == 0:
        print("\n## No output files at this path; unable to create frequency dictionary. ##\n")
        return 0
    # Check for pre-existing dictionary file or other non-coherence data files and remove them from list
    excise_list = []
    for channel in channel_list:
        if "dictionary_signif_coherences_" in channel:
            excise_list.append(channel)
            print("Pre-existing dictionary file found. This file will be over-written:" + channel +"\n")
        elif channel[0:18] != "signif_coherences_":
            excise_list.append(channel)
```

```
265.            print("Non-
    coherence data file found, removing from analysis: " + channel + "\n")
266.    for channel in excise_list:
267.        channel_list.remove(channel)
268.    print("Number of channels in dictionary: " + str(len(channel_list)))
269.
270.    # Initialize frequency dictionary
271.    dict = {}
272.
273.    for file in channel_list[:]: #Iterating over every file in directory to get coherence
    data
274.        print("Adding to dictionary: " + file)
275.        #Finding high-coherence data
276.        freqdataline = subprocess.check_output('grep -
    n "## Format: \[frequency\] \[coherence\] ##" ' + file
277.                                               + " | awk '{print $1}'", shell=True).rstrip
    ()
278.        freqdataline = int(freqdataline[:-3])
279.
280.        chdata = open(file, "r")
281.        line_index = 0
282.        for line in chdata:
283.            if line_index >= freqdataline:
284.                pair = line.split()
285.                pair = [float(value) for value in pair]
286.                freq = pair[0]
287.
288.                if freq not in dict: #Add new frequency to dictionary
289.                    dict[freq] = [file]
290.
291.                elif freq in dict: #Add this channel to list value for this frequency key
    in dictionary
292.                    getlist = dict[freq]
293.                    getlist.append(file)
294.                    dict.update({freq: getlist})
295.
296.            line_index += 1
297.        chdata.close()
298.
299.    freqlist = dict.keys()
300.    freqlist.sort()
301.
302.    # Finding date of fscan via output files this function has been directed to look for
303.    fscan_date = subprocess.check_output('grep -n "Date of fscans: " ' + channel_list[0]
304.                                         + " | awk '{print $4}'", shell=True).rstrip()
305.
306.    # Writing the dictionary output file for this channel
307.    with open("dictionary_signif_coherences_" + duration + fscan_date + ".txt", "w") as ou
    tputfile:
308.        outputfile.write("Frequency dictionary for significantly coherent channels\n")
309.        outputfile.write("Type of fscans: " + duration[:-1] + "\n")
310.        outputfile.write("Date of fscans: " + fscan_date + "\n")
311.        outputfile.write("Dictionary generated from output path: " + outputpath + "\n")
312.        outputfile.write("Number of unique channels in dictionary: " + str(len(channel_lis
    t)) + "\n")
313.        outputfile.write("\nBased on provided arguments, this dictionary output:\n")
314.        if output_cutoff:
315.            outputfile.write("- only lists band information, and does NOT contain a full \
    n"
316.                             "  frequency dictionary below the list of bands (argument out
    put_cutoff == 'y')\n")
```

```
317.        else:
318.            outputfile.write("- contains more detailed frequency dictionary information\n"
319.                             "  below the list of bands (argument output_cutoff == 'n')\n"
    )
320.        if not include_60Hz_band:
321.            outputfile.write("- WILL NOT list the 60Hz band, even if it was significant (a
    rgument include_60Hz_band == 'n')\n")
322.        else:
323.            outputfile.write("- WILL list the 60Hz band, if it was significant (argument i
    nclude_60Hz_band == 'y')\n")
324.        if not include_integer:
325.            outputfile.write("- WILL NOT list bands containing integer frequencies,\n"
326.                             "  even if those bands were significant (argument include_int
    eger == 'n')\n")
327.        else:
328.            outputfile.write("- WILL list bands containing integer frequencies,\n"
329.                             "  if those bands were significant (argument include_integer
    == 'y')\n")
330.
331.        outputfile.write("\nCriteria for two frequencies to be considered in a band is f1
    - f2 <= {0} Hz.\n"
332.                         "This corresponds to a maximum of {1} frequency bins apart (argum
    ent band_cutoff == {1})."
333.                         "\n".format(freq_cutoff, band_cutoff))
334.
335.        #For reference: channel_info_dict = {get_chan_name(coherencefilename): [signifcohe
    rence, signifcohlist], ...}
336.        threshold_list = []
337.        for channel in channel_info_dict:
338.            threshold_list.append(channel_info_dict[channel][0])
339.
340.        outputfile.write("\nThe average significant coherence threshold for these channels
     was {0}.\n"
341.                         "The lowest significant coherence threshold was {1}.\n"
342.                         "The highest significant coherence threshold was {2}.\n"
343.                         "".format(sum(threshold_list)/len(threshold_list), min(threshold_
    list), max(threshold_list)))
344.
345.        # Write out the band summary of the frequency dictionary
346.        # Possible future idea: [channel 1] [number of high-
    coherence frequencies of channel 1 within band]
347.
348.        outputfile.write("\n## Frequency Dictionary Band Summary ##\n"
349.                         "## Format: ##\n"
350.                         "# [lowest band frequency] [highest band frequency] [bandwidth]\n
    "
351.                         "    [channel 1] [max channel 1 coherence in band]\n"
352.                         "    [channel 2] [max channel 2 coherence in band]\n"
353.                         "    (etc.)\n")
354.
355.        # Finding bands of interest in highly coherent data
356.        templist = []  # Initialize list for given band
357.        # print("\nCoherence band (templist) format: [freqlo] [freqhi] [chan1], [chan2], .
    ..")
358.
359.        for row in range(len(freqlist) - 1):
360.            freq = freqlist[row]
361.            nextfreq = freqlist[row + 1]
362.
363.            if row == 0:  # Initialize first frequency in band summary
```

```
364.                templist = [freq, freq]
365.                for file in dict[freq]:
366.                    templist.append(get_chan_name(file))
367.
368.            if nextfreq - freq <= (freq_cutoff + 0.0001):  # If next line is within same f
    requency band
369.                for file in dict[nextfreq]:
370.                    if get_chan_name(file) not in templist:
371.                        templist.append(get_chan_name(file))
372.
373.                if row == len(freqlist) - 2:  # Handle case where next row is last row
374.                    templist[1] = nextfreq
375.                    #print("#### FINAL NEW BAND ####")
376.                    #print(templist)
377.                    write_dict_entry(outputfile, templist, include_60Hz_band, include_inte
    ger, channel_info_dict)
378.
379.            else:  # If not within same band
380.                templist[1] = freq
381.                #print("#### NEW BAND ####")
382.                #print(templist)
383.                write_dict_entry(outputfile, templist, include_60Hz_band, include_integer,
     channel_info_dict)
384.
385.                templist = [nextfreq, nextfreq]  # Prep templist for new band
386.                for file in dict[nextfreq]:
387.                    if get_chan_name(file) not in templist:
388.                        templist.append(get_chan_name(file))
389.
390.                if row == len(freqlist) - 2:  # Handle case where next row is last row
391.                    write_dict_entry(outputfile, templist, include_60Hz_band, include_inte
    ger, channel_info_dict)
392.
393.        if not output_cutoff:
394.            # Write out the full dictionary one frequency at a time
395.            outputfile.write("\n## Frequency Dictionary ##\n"
396.                             "## Format: ##\n"
397.                             "# [frequency]\n"
398.                             "    [channel 1]\n"
399.                             "    [channel 2]\n"
400.                             "    (etc.)\n")
401.            for freq in freqlist:
402.                outputfile.write("\n# {0:6f}\n".format(freq))
403.                for file in dict[freq]:
404.                    chan_name = get_chan_name(file)
405.                    outputfile.write("    " + chan_name + "\n")
406.
407.    outputfile.close()
408.
409.    print("\nDictionary complete.")
410.
411.    return None
412.
413.##############
414.# Main Code #
415.##############
416.def main():
417.    initpath = subprocess.check_output("pwd").rstrip()
418.    print("\n## Starting execution at: " + str(datetime.now())[0:19] + " ##")
419.    print("Start working directory: " + initpath + "\n")
420.
```

```
421.     parser = argparse.ArgumentParser(
422.         description = "Find Fscan coherence data and identify unusually high coherences. \
    n"
423.                       "Complexity of output remains under construction.")
424.
425.     parser.add_argument("targetpath", type=str,
426.                         help="the path of the directory of the fscan coherence data to be
    analyzed")
427.     parser.add_argument("outputpath", type=str,
428.                         help="the output directory path, where results will be written to"
    )
429.     parser.add_argument("fscantype", type=str,
430.                         help="type of fscan: m (monthly), w (weekly), or d (daily)")
431.     parser.add_argument("dodict", type=str,
432.                         help="make a freq range dictionary, y (yes) or n (no)")
433.     parser.add_argument("output_cutoff", type=str,
434.                         help="cut out certain data to abbreviate output, y (yes) or n (no)
    ")
435.     parser.add_argument("include_integer", type=str,
436.                         help="include integer frequencies in the output, y (yes) or n (no)
    ")
437.     parser.add_argument("include_60Hz_band", type=str,
438.                         help="include any frequency band containing 60 Hz in the dictionar
    y, y (yes) or n (no)")
439.     parser.add_argument("significance_coefficient", type=float,
440.                         help="whole number on the order of 1-
    10 to use in calculation of significant coherence; \n"
441.                              "denotes the power of 10 to use for the coefficient; \n"
442.                              "standard is 1 (makes coefficient 10 ** 1 = 10); \n"
443.                              "higher values generate higher significant coherences and sho
    rter outputs; \n"
444.                              "smaller values generate lower significant coherences and lon
    ger outputs")
445.     parser.add_argument("band_cutoff", type=int,
446.                         help="a whole number that denotes the max number of frequency bins
     between "
447.                              "highly coherent frequencies to place them in the same freque
    ncy band")
448.
449.     args = parser.parse_args() #Getting the paths, double-check their validity
450.     targetpath = args.targetpath.rstrip()
451.     outputpath = args.outputpath.rstrip()
452.     fscantype = args.fscantype.rstrip()
453.     dodict = args.dodict.rstrip()
454.     output_cutoff = args.output_cutoff.rstrip()
455.     include_integer = args.include_integer.rstrip()
456.     include_60Hz_band = args.include_60Hz_band.rstrip()
457.     significance_coefficient = float(args.significance_coefficient)
458.     band_cutoff = args.band_cutoff
459.
460.     assert os.path.exists(targetpath), "Unable to locate target path."
461.     assert os.path.exists(outputpath), "Unable to locate output path."
462.     assert fscantype == "m" or fscantype == "w" or fscantype == "d", "Must specify type of
     fscan: " \
463.                                                                     "m (monthly), w (weekly),
    or d (daily)."
464.     assert dodict == "y" or dodict == "n", "Must specify if doing a dictionary: y (yes) or
     n (no)."
465.     assert output_cutoff == "y" or output_cutoff == "n", "Must specify if abbreviating out
    put: y (yes) or n (no)."
```

```
    assert include_integer == "y" or include_integer == "n", "Must specify if including integer frequencies: y (yes) or n (no)."
    assert include_60Hz_band == "y" or include_60Hz_band == "n", "Must specify if including 60Hz band: y (yes) or n (no)."
    assert type(band_cutoff) == int, "Band cutoff criteria must be a whole number of frequency bins"
    assert band_cutoff > 0, "Band cutoff criteria must be a whole number of frequency bins"

    if fscantype == "d":
        duration = "daily_"
    elif fscantype == "w":
        duration = "weekly_"
    elif fscantype == "m":
        duration = "monthly_"
    assert duration == "monthly_" or duration == "weekly_" or duration == "daily_", \
        "Error in recognizing fscan type (daily, weekly, monthly)"

    if output_cutoff == "y":
        output_cutoff = True
    else:
        output_cutoff = False

    if include_integer == "y":
        include_integer = True
    else:
        include_integer = False

    if include_60Hz_band == "y":
        include_60Hz_band = True
    else:
        include_60Hz_band = False

    channel_list = get_channel_list(targetpath)

    """
    Note: the default SFT length for Fscans is T = 1800 seconds.
    The frequency resolution is 1/T = 1/1800 Hz =(rounded) 0.0005555 Hz = 5.5555*10^(-4) Hz.
    So at the most granular level, frequency bins are separated by 0.0005555 Hz.
    """
    # band_cutoff specifies number of frequency bins, freq_cutoff translates this into a frequency bandwidth
    freq_cutoff = (1.0/1800.0 * float(band_cutoff))
    #print(band_cutoff)
    #print(freq_cutoff)
    channel_info_dict = {}

    for channel in channel_list:
        chanpath = targetpath + "/" + channel
        attempt = analyze_channel(chanpath, outputpath, duration, significance_coefficient, include_integer, freq_cutoff)
        if type(attempt) == dict:
            channel_info_dict.update(attempt)

    if dodict == "y":
        makedict(outputpath, duration, output_cutoff, freq_cutoff, band_cutoff, include_60Hz_band, include_integer, channel_info_dict)

    endpath = subprocess.check_output("pwd").rstrip()
    print("\nEnd working directory: " + endpath)
```

```
519.     print("## Ending execution at: " + str(datetime.now())[0:19] + " ##\n")
520. 
521. if __name__ == "__main__":
522.     main()
```

Appendix B: `script_modmultidayfscan.py`

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
@author: Thomas Harris <harristr@whitman.edu>
"""

__author__ = 'Thomas Harris <harristr@whitman.edu>'

"""
This script has two immediately related purposes:
- write .rsc files for running Fscan using a modified form of multiFscanGenerator.tcl
  (multiFscanGenerator.tcl takes its parameters from specified .rsc file)
- run command line arguments at various time intervals to gradually create batches of Fscan jobs
"""

#Importing useful tools

import os
import subprocess
from datetime import datetime
import re
import time

##############
# Functions #
##############

def make_new_rsc_file(rscname, rsc_lines, gpsstart, gpsend):
    """
    Makes a new .rsc file using an open file object as a template.
    Replaces the start and end GPS times in the .rsc with new times.
    :param rscname: string name of the template .rsc file being opened
    :param rscfile: list of lines of template .rsc file for Fscans to use
    :param gpsstart: starting GPS time of Fscans
    :param gpsend: ending GPS time of Fscans
    :return:
    """

    #setting up to search for start and end GPS times in the template .rsc file
    re_start = re.compile('set startTime "\d{10}";')
    start_found = False
    re_end = re.compile('set endTime "\d{10}";')
    end_found = False
    line_number = 0

    #Checking to see if GPS start and end times are properly set in template .rsc file
    for line in rsc_lines:
        #if 25 < line_number < 29:
        #    print(line.rstrip())
        if re_start.match(line):
            start_found = True
            #print("Start GPS time on line number " + str(line_number))
        elif re_end.match(line):
            end_found = True
```

```
            #print("End GPS time on line number " + str(line_number))
        line_number += 1

    assert start_found, "Starting GPS time not properly found in template .rsc file"
    assert end_found, "Ending GPS time not properly found in template .rsc file"

    #Creating name and file for new .rsc file based on template
    new_rsc_name = "{0}_{1}_{2}.rsc".format(rscname[:-4], gpsstart, gpsend)
    with open(new_rsc_name, "w") as new_rsc:
        for line in rsc_lines:
            if re_start.match(line):
                new_rsc.write('set startTime "{0}";\n'.format(gpsstart))
            elif re_end.match(line):
                new_rsc.write('set endTime "{0}";\n'.format(gpsend))
            else:
                new_rsc.write(line)

    return new_rsc_name

##############
# Main Code #
##############

def main():
    initpath = subprocess.check_output("pwd").rstrip()
    print("\n## Starting execution at: " + str(datetime.now())[0:19] + " ##")
    print("Start working directory: " + initpath + "\n")

    subprocess.check_output("cd /home/thomas.harris/public_html/fscan/test", shell=True).r
strip()
    print("Now in directory: " + subprocess.check_output("pwd", shell=True).rstrip() + "\n
")

    rscname = "myFscansH1L1.rsc"

    seconds_per_day = 86400
    start_time = 1261875620
    sleep_time = 3600 #Number of seconds to sleep for between starting Fscan jobs on diffe
rent days
    number_days = 31 #Number of days to run Fscans on following start_time

    assert type(start_time) == int, "Starting GPS time must be an integer"
    assert type(number_days) == int, "Must use integer number of days"
    assert rscname[-4:] == ".rsc", "Resource file name must end in extension '.rsc'"

    gps_day_list = []
    day_start = start_time

    #Making a list of GPS time pairs for writing new .rsc files with these GPS times
    for day in range(number_days):
        day_end = day_start + seconds_per_day
        gps_day_list.append([day_start, day_end])
        day_start = day_end

    rsc_list = []

    #Generate all of the .rsc files and put their names into a list
    with open(rscname, "r") as rsc_file:
        rsc_lines = rsc_file.readlines()
        for day in gps_day_list:
            rsc_list.append(make_new_rsc_file(rscname, rsc_lines, day[0], day[1]))
```

```
113.
114.     print("New .rsc files generated:")
115.     for file_name in rsc_list:
116.         print(file_name)
117.     print("\n")
118.
119.     if os.path.exists("./lastTimeUsedByH1FscanCoherenceGeneratorAuto.txt"):
120.         print("Removing previous lastTimeUsedByH1FscanCoherenceGeneratorAuto.txt file.")
121.         print(subprocess.check_output("rm lastTimeUsedByH1FscanCoherenceGeneratorAuto.txt"
    , shell=True).rstrip())
122.
123.     #Run Fscans using the .rsc files and delay between starting each Fscan generation
124.     for file_name in rsc_list:
125.         run_fscan_command = "./modH1L1_multiFscanGenerator.tcl ./{0} -
    R".format(file_name)
126.         print("Running Fscans using " + file_name + " at " + str(datetime.now())[0:19] + "
    .")
127.         #print(run_fscan_command)
128.         print(subprocess.check_output(run_fscan_command, shell=True).rstrip())
129.
130.         if file_name != rsc_list[-1]:
131.             print("Current time is " + str(datetime.now())[0:19] + ".")
132.             print("Sleeping for {0} seconds.\n".format(sleep_time))
133.             time.sleep(sleep_time)
134.             if os.path.exists("./lastTimeUsedByH1FscanCoherenceGeneratorAuto.txt"):
135.                 print("Removing previous lastTimeUsedByH1FscanCoherenceGeneratorAuto.txt f
    ile.")
136.                 print(subprocess.check_output("rm lastTimeUsedByH1FscanCoherenceGeneratorA
    uto.txt", shell=True).rstrip())
137.
138.     endpath = subprocess.check_output("pwd").rstrip()
139.     print("\nEnd working directory: " + endpath)
140.     print("## Ending execution at: " + str(datetime.now())[0:19] + " ##\n")
141.
142. if __name__ == "__main__":
143.     main()
```

Appendix C: segment_intersector.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
@author: Thomas Harris <harristr@whitman.edu>
"""

__author__ = 'Thomas Harris <harristr@whitman.edu>'

#Importing useful tools

import argparse
import sys
import os
import glob
import subprocess
from datetime import datetime

##############
# Functions #
##############

def get_segments(ifo, gpsstarttime, gpsendtime):
    """
    Uses ligolw_segment_query_dqsegdb to get DMT-ANALYSIS_READY segments
    :param ifo: H1 or L1
    :param gpsstarttime: int to go after --gps-start-time
    :param gpsendtime: int to go after --gps-end-time
    :return:
    """
    print("Retrieving " + ifo + " segments.")
    tempsegfile = "{0}segs_{1}_{2}.txt".format(ifo, gpsstarttime, gpsendtime)
    print("Name of temporary segment file: {0}".format(tempsegfile))
    subprocess.check_output('ligolw_segment_query_dqsegdb --segment-
url=https://segments.ligo.org --query-segments '
                            '--include-segments {0}:DMT-ANALYSIS_READY:1 --gps-start-
time {1} --gps-end-time {2} '
                            '| /bin/grep -v "0, 0" | /usr/bin/ligolw_print -
t segment:table -c start_time -c end_time '
                            '-
d " " > {3}'.format(ifo,gpsstarttime,gpsendtime,tempsegfile), shell=True).rstrip()
    return tempsegfile

##############
# Main Code #
##############
def main():
    initpath = subprocess.check_output("pwd").rstrip()
    print("\n## Starting execution at: " + str(datetime.now())[0:19] + " ##")
    print("Start working directory: " + initpath + "\n")

    parser = argparse.ArgumentParser(
        description = "This script is used by a modified version of the multiFscanGenerato
r.tcl \n"
                      "to generate Fscan outputs between interferometers (H1 and L1).\n"
                      "Intersects the DMT-ANALYSIS_READY:1 segments from H1 and L1.")
```

```
51.
52.     parser.add_argument("gpsstart", type=int,
53.                         help="starting GPS time")
54.     parser.add_argument("gpsend", type=int,
55.                         help="ending GPS time")
56.
57.     args = parser.parse_args() #Getting the paths, double-check their validity
58.     gpsstart = args.gpsstart
59.     gpsend = args.gpsend
60.
61.     assert type(gpsstart) == int, "Starting GPS time must be an integer"
62.     assert type(gpsend) == int, "Ending GPS time must be an integer"
63.
64.     #Demo GPS times: (1265331620, 1265418019)
65.     h1segs = get_segments("H1", gpsstart, gpsend)
66.     l1segs = get_segments("L1", gpsstart, gpsend)
67.     #print(h1segs)
68.     #print(l1segs)
69.
70.     print("Intersecting H1:DMT-ANALYSIS_READY:1 and L1:DMT-ANALYSIS_READY:1 segments.")
71.
72.     pwd = subprocess.check_output("pwd", shell=True).rstrip()
73.     #h1Segfile = "{0}/{1}".format(pwd, h1segs)
74.     #l1Segfile = "{0}/{1}".format(pwd,l1segs)
75.     #intersectOutputfile = "{0}/H1L1Intersectedsegs_{1}_{2}.txt".format(pwd, gpsstart, gps
    end)
76.     #h1Segfile = "H1segs_1265331620_1265418019.txt"
77.     #l1Segfile = "L1segs_1265331620_1265418019.txt"
78.     h1Segfile = h1segs
79.     l1Segfile = l1segs
80.     intersectOutputfile = "{0}/H1L1Intersectedsegs_{1}_{2}.txt".format(pwd,gpsstart, gpsen
    d)
81.     segexprCommand = "segexpr \"intersection(%s,%s)\" %s" %(h1Segfile, l1Segfile, intersec
    tOutputfile)
82.     subprocess.check_output(segexprCommand, shell=True).rstrip()
83.     print("Intersected segment file: " + intersectOutputfile)
84.
85.     endpath = subprocess.check_output("pwd").rstrip()
86.     print("\nEnd working directory: " + endpath)
87.     print("## Ending execution at: " + str(datetime.now())[0:19] + " ##\n")
88.
89. if __name__ == "__main__":
90.     main()
```

Appendix D: `fscan_lines.py`

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
@author: Thomas Harris <harristr@whitman.edu>
"""

__author__ = 'Thomas Harris <harristr@whitman.edu>'

#Importing useful tools

import argparse
import sys
import os
import subprocess
from datetime import datetime
from scipy import stats
import numpy

##############
# Functions #
##############

def get_channel_list(date):
    """
    :param date: string saying what dates to focus on
    :return: List containing strings of directory names for analysis
    """
    channel_list = subprocess.check_output("ls | grep 'fscans_{0}_.*' | cat".format(date), shell=True).split()
    #print("Number of days included in this analysis: {0}\n".format(str(len(channel_list))))
    return channel_list

def analyze_channel(chanpath, probability_cutoff):
    """
    :param chanpath: path to channel for analysis
    :param outputpath: float cutoff probability criteria for high-power lines
    :return:
    """
    os.chdir(chanpath)  # Going to the target directory
    print("Current directory: " + subprocess.check_output("pwd").rstrip() + "")

    # Find power spectrum data file
    spectrumfile = subprocess.check_output("ls -1 spec_0.00_100.00*.txt | grep -v -E '*combs*|*sorted*'", shell=True).rstrip()
    #print("Spectrum file name: {0}".format(spectrumfile))

    with open(spectrumfile, "r") as file:
        data = file.readlines()

    data = data[1:]
    for line in range(len(data)):
        data[line] = float(data[line].split()[1])
    print("Spectrum data obtained from {0}.".format(spectrumfile))

    """
    print(len(data))
    for i in range(10):
        print(data[i])
        print(type(data[i]))
```

```
    """
    # Find number of averages used in coherence calculations:
    # First check that the coherence data exists in this frequency range, fail if not.

    prelim = subprocess.check_output("ls -
l logs/MakeSFTs*.out | awk '{print $NF}'", shell=True).rstrip()
    if prelim == None or prelim == "":
        print("\n## ## ## LINE ANALYSIS FAILED FOR THIS DAY. ## ## ##\n")
        return 1
    prelim = prelim.split()
    deg_of_freedom = len(prelim) * 2
    print("Degrees of freedom = 2 * number of SFTs = 2 * number of averages : {0}".format(
deg_of_freedom))
    #NOTE: Degrees of freedom = 2 * numSFTs = 2 * number of averages

    #print(stats.chi2.isf(probability_cutoff, deg_of_freedom))
    cutoff = float(stats.chi2.isf(probability_cutoff, deg_of_freedom)) / float(deg_of_free
dom)
    print("Normalized threshold power = {0}".format(cutoff))
    count = 0
    for num in data:
        if num >= cutoff:
            count += 1
    print("Number of high power lines: {0}\n".format(count))
    return count

##############
# Main Code #
##############
def main():
    print("\n## Starting execution at: " + str(datetime.now())[0:19] + " ##")
    print("Start working directory: " + subprocess.check_output("pwd").rstrip() + "\n")

    parser = argparse.ArgumentParser(
        description = "Find lines in Fscan spectrum data. \n"
                      "Uses chi2 statistics to find a threshold for statistically signific
ant "
                      "power levels in the spectra, and identifies days with unusually hig
h number of lines.")

    parser.add_argument("targetpath", type=str,
                        help="the path of the directory of daily fscan coherence data to b
e analyzed, e.g. "
                             "'/home/pulsar/public_html/fscan/H1/daily/H1Fscan_coherence/H
1Fscan_coherence'")
    """
    parser.add_argument("outputpath", type=str,
                        help="the output directory path, where results will be written to"
)
    """
    parser.add_argument("date", type=str,
                        help="month to be analyzed, formatted as yyyy_mm (e.g. '2020_02')"
)
    parser.add_argument("probability_cutoff", type=float,
                        help="cutoff probability criteria for high-power lines")

    args = parser.parse_args() #Getting the paths, double-check their validity
    targetpath = args.targetpath.rstrip()
    #outputpath = args.outputpath.rstrip()
    date = args.date.rstrip()
    probability_cutoff = args.probability_cutoff

    print("Fscan spectra line finding script with chi2 statistics\n"
          "Date target for this line counter: {0}\n"
          "Cutoff probability criteria for high-
power lines: {1}\n".format(date, probability_cutoff))

    os.chdir(targetpath)  # Going to the target directory
    current_chan_path = subprocess.check_output("pwd").rstrip()
```

```
117.    print("Current working directory: {0}\n".format(current_chan_path))
118.
119.    date_list = get_channel_list(date)
120.
121.    if targetpath[-1] == "/":
122.        targetpath = targetpath[:-1]
123.
124.    total_list = []
125.    for day in date_list:
126.        try:
127.            total_list.append([day, analyze_channel("{0}/{1}/H1_GDS-
    CALIB_STRAIN".format(targetpath,day), probability_cutoff)])
128.        except:
129.            print("## LINE ANALYSIS FAILED FOR: {0} ##\n".format(day))
130.
131.    count_list = []
132.    for pair in total_list:
133.        count_list.append(pair[1])
134.
135.    stdev = numpy.std(count_list)
136.
137.    print("\nNumber of days with available data: {3}"
138.          "\nAverage number of daily high-power lines in H1_GDS-
    CALIB_STRAIN during {0}: {1}"
139.          "\nStandard deviation in number of daily high-power lines: {2}"
140.          "".format(date, numpy.mean(count_list), stdev, len(total_list)))
141.
142.    print("\nHigh line density days during {0}:".format(date))
143.    for pair in total_list:
144.        if pair[1] >= numpy.mean(count_list) + 2 * stdev:
145.            print("{0}, number of high-power lines = {1}".format(pair[0], pair[1]))
146.
147.    print("\nEnd working directory: " + subprocess.check_output("pwd").rstrip())
148.    print("## Ending execution at: " + str(datetime.now())[0:19] + " ##\n")
149.
150.if __name__ == "__main__":
151.    main()
```

Appendix E: Sample Coherence Dictionary

```
Frequency dictionary for significantly coherent channels
Type of fscans: monthly
Date of fscans: fscans_2020_03_01_04_00_03_PST_Sun
Dictionary generated from output path:
/home/thomas.harris/public_html/fscan/sample_outputs/script_v4_samples/short_full_month_sample_202
0_03/
Number of unique channels in dictionary: 42

Based on provided arguments, this dictionary output:
- only lists band information, and does NOT contain a full
  frequency dictionary below the list of bands (argument output_cutoff == 'y')
- WILL NOT list the 60Hz band, even if it was significant (argument include_60Hz_band == 'n')
- WILL NOT list bands containing integer frequencies,
  even if those bands were significant (argument include_integer == 'n')

Criteria for two frequencies to be considered in a band is f1 - f2 <= 0.00277777777778 Hz.
This corresponds to a maximum of 5 frequency bins apart (argument band_cutoff == 5).

The average significant coherence threshold for these channels was 0.0186103583995.
The lowest significant coherence threshold was 0.0186103583995.
The highest significant coherence threshold was 0.0186103583995.

## Frequency Dictionary Band Summary ##
## Format: ##
# [lowest band frequency] [highest band frequency] [bandwidth]
    [channel 1] [max channel 1 coherence in band]
    [channel 2] [max channel 2 coherence in band]
    (etc.)

# 0.599444 0.600556 0.001112
    H1_PEM-EX_MAG_EBAY_SEIRACK_X_DQ 0.0393
    H1_PEM-EY_MAG_EBAY_SEIRACK_X_DQ 0.0388
    H1_PEM-EY_MAG_EBAY_SEIRACK_Y_DQ 0.0342
    H1_PEM-EY_MAG_EBAY_SEIRACK_Z_DQ 0.0308
    H1_PEM-EY_MAG_EBAY_SUSRACK_Y_DQ 0.0324
    H1_PEM-EX_MAG_EBAY_SEIRACK_Y_DQ 0.0255
    H1_PEM-EX_MAG_EBAY_SEIRACK_Z_DQ 0.0251
    H1_PEM-EY_MAG_EBAY_SUSRACK_Z_DQ 0.0205

# 1.600000 1.600556 0.000556
    H1_PEM-EX_MAG_EBAY_SEIRACK_X_DQ 0.019

# 1.900000 1.900000 0.000000
    H1_PEM-EX_MAG_EBAY_SEIRACK_X_DQ 0.0235

# 3.235000 3.235000 0.000000
    H1_PEM-EX_MAG_EBAY_SUSRACK_Y_DQ 0.0192

# 3.260000 3.260000 0.000000
    H1_PEM-EX_MAG_EBAY_SUSRACK_Y_DQ 0.021

# 3.264444 3.268333 0.003889
    H1_PEM-EX_MAG_EBAY_SUSRACK_Y_DQ 0.0213

# 3.274444 3.274444 0.000000
    H1_PEM-EX_MAG_EBAY_SUSRACK_Y_DQ 0.0238

# 3.283333 3.283889 0.000556
    H1_PEM-EX_MAG_EBAY_SUSRACK_Y_DQ 0.0233

# 3.289444 3.290556 0.001112
    H1_PEM-EX_MAG_EBAY_SUSRACK_Y_DQ 0.0265
```

```
# 3.293889 3.333333 0.039444
    H1_PEM-EX_MAG_EBAY_SUSRACK_Y_DQ 0.0408
    H1_PEM-CS_MAG_EBAY_LSCRACK_X_DQ 0.0214

# 3.336667 3.662222 0.325555
    H1_PEM-EX_MAG_EBAY_SUSRACK_Y_DQ 0.0511
    H1_PEM-CS_MAG_EBAY_LSCRACK_X_DQ 0.2604
    H1_PEM-EX_MAG_EBAY_SUSRACK_X_DQ 0.028
    H1_PEM-CS_MAG_EBAY_LSCRACK_Y_DQ 0.0208
    H1_PEM-CS_MAG_EBAY_LSCRACK_Z_DQ 0.0441
    H1_PEM-EY_MAG_EBAY_SUSRACK_Y_DQ 0.0216
    H1_PEM-EX_SEIS_VEA_FLOOR_Z_DQ 0.0253

# 3.666667 3.674444 0.007777
    H1_PEM-EX_MAG_EBAY_SUSRACK_Y_DQ 0.0187
    H1_PEM-CS_MAG_EBAY_LSCRACK_X_DQ 0.037

[ FILE BREAK INSERTED ]

# 11.370560 11.370560 0.000000
    H1_PEM-EX_MAG_EBAY_SUSRACK_Z_DQ 0.0218

# 11.380560 11.386110 0.005550
    H1_PEM-EX_MAG_EBAY_SUSRACK_Z_DQ 0.0252

# 11.393890 11.395560 0.001670
    H1_PEM-EX_MAG_EBAY_SEIRACK_X_DQ 0.0509
    H1_PEM-EX_MAG_EBAY_SEIRACK_Z_DQ 0.0547
    H1_PEM-EX_MAG_EBAY_SUSRACK_X_DQ 0.0263
    H1_PEM-EX_MAG_EBAY_SUSRACK_Y_DQ 0.0546
    H1_PEM-EX_MAG_EBAY_SUSRACK_Z_DQ 0.0436
    H1_PEM-EX_MAG_VEA_FLOOR_Y_DQ 0.034
    H1_PEM-EX_TILT_VEA_FLOOR_T_DQ 0.0608
    H1_PEM-EX_MAG_EBAY_SEIRACK_Y_DQ 0.0213

# 11.422780 11.422780 0.000000
    H1_PEM-EX_MAG_EBAY_SUSRACK_Z_DQ 0.0198

# 11.436670 11.440000 0.003330
    H1_PEM-EX_MAG_EBAY_SUSRACK_Z_DQ 0.0197

[ FILE BREAK INSERTED ]

# 99.646670 99.646670 0.000000
    H1_PEM-EX_MAG_EBAY_SUSRACK_X_DQ 0.0222
    H1_PEM-EX_MAG_EBAY_SUSRACK_Z_DQ 0.0198

# 99.744440 99.744440 0.000000
    H1_PEM-EX_TILT_VEA_FLOOR_T_DQ 0.0225

# 99.958330 99.993330 0.035000
    H1_PEM-EY_MAG_EBAY_SEIRACK_Y_DQ 0.4767
    H1_PEM-EY_MAG_EBAY_SUSRACK_X_DQ 0.4749
    H1_PEM-EY_MAG_EBAY_SUSRACK_Y_DQ 0.4726
    H1_PEM-EY_MAG_EBAY_SUSRACK_Z_DQ 0.477
    H1_PEM-EY_MAG_EBAY_SEIRACK_X_DQ 0.4755
    H1_PEM-EY_MAG_EBAY_SEIRACK_Z_DQ 0.4721
    H1_PEM-EY_TILT_VEA_FLOOR_T_DQ 0.45
    H1_PEM-EX_MAG_EBAY_SEIRACK_X_DQ 0.8447
    H1_PEM-EX_MAG_EBAY_SEIRACK_Z_DQ 0.8375
    H1_PEM-EX_MAG_EBAY_SUSRACK_X_DQ 0.8148
    H1_PEM-EX_MAG_EBAY_SUSRACK_Y_DQ 0.8337
    H1_PEM-EX_MAG_EBAY_SUSRACK_Z_DQ 0.8433
    H1_PEM-EX_TILT_VEA_FLOOR_T_DQ 0.8386
    H1_PEM-EX_MAG_EBAY_SEIRACK_Y_DQ 0.5699
    H1_PEM-EX_MAG_VEA_FLOOR_Y_DQ 0.7116
    H1_PEM-EY_MAG_VEA_FLOOR_X_DQ 0.0996
    H1_PEM-EY_MAG_VEA_FLOOR_Z_DQ 0.0784
    H1_PEM-EY_TILT_VEA_FLOOR_Y_DQ 0.1661
```

```
    H1_PEM-EX_MAG_VEA_FLOOR_Z_DQ 0.5674
    H1_PEM-EX_MAG_VEA_FLOOR_X_DQ 0.4199

# 99.998330 99.998890 0.000560
    H1_PEM-CS_MAG_EBAY_LSCRACK_X_DQ 0.0707
    H1_PEM-CS_MAG_EBAY_LSCRACK_Y_DQ 0.0715
    H1_PEM-CS_MAG_EBAY_LSCRACK_Z_DQ 0.0712
    H1_PEM-CS_MAG_LVEA_VERTEX_Y_DQ 0.0703
    H1_PEM-CS_TILT_LVEA_VERTEX_T_DQ 0.0721
    H1_PEM-CS_TILT_LVEA_VERTEX_Y_DQ 0.0395
    H1_PEM-EX_MAG_EBAY_SEIRACK_Y_DQ 0.0529
    H1_PEM-EX_MAG_EBAY_SEIRACK_Z_DQ 0.0437
    H1_PEM-EY_MAG_EBAY_SEIRACK_Y_DQ 0.052
    H1_PEM-EY_MAG_EBAY_SEIRACK_Z_DQ 0.0633
    H1_PEM-CS_MAG_LVEA_VERTEX_Z_DQ 0.0195
    H1_PEM-CS_TILT_LVEA_VERTEX_X_DQ 0.0224
    H1_PEM-EX_MAG_EBAY_SEIRACK_X_DQ 0.0187
    H1_PEM-EY_MAG_EBAY_SEIRACK_X_DQ 0.0279
```

Appendix F: Sample H1L1 Coherence Data

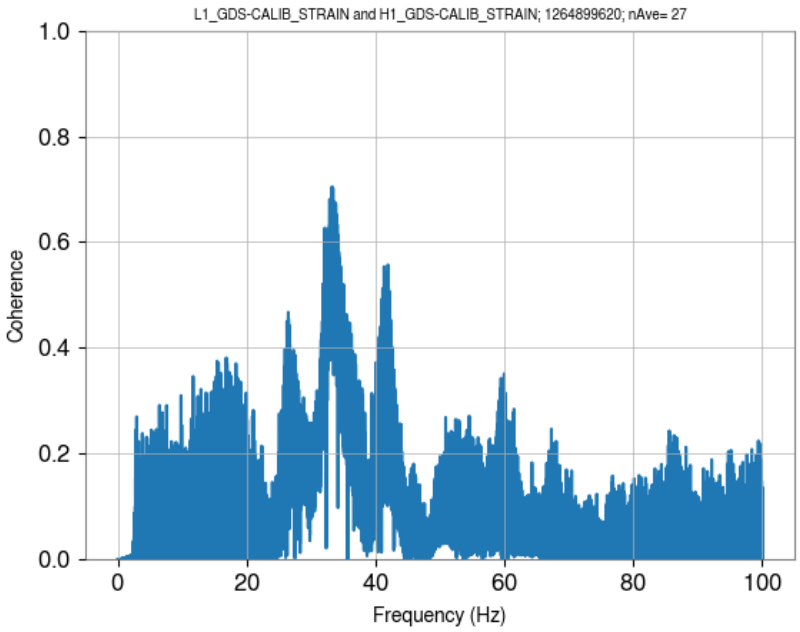

Figure 14: Sample Hanford-Livingston daily coherence data; Feb. 5, 2020 [20]

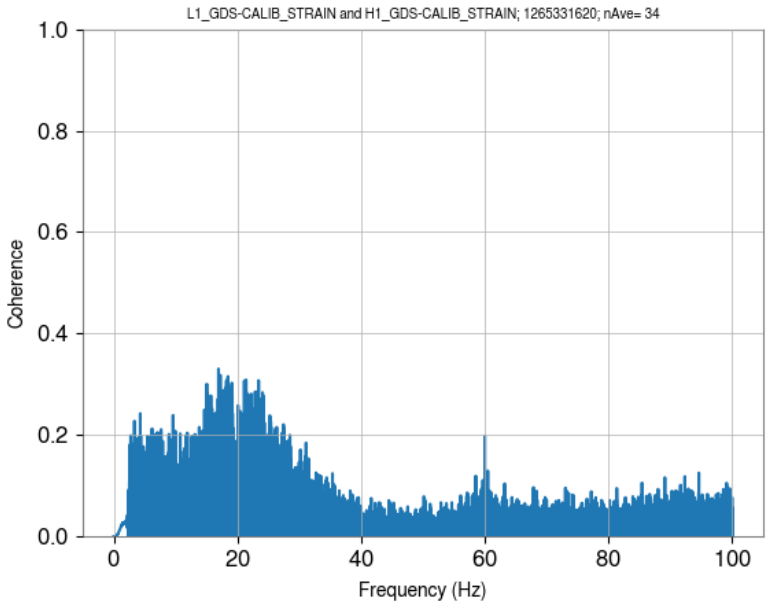

Figure 15: Sample Hanford-Livingston daily coherence data; Feb. 10, 2020 [20]

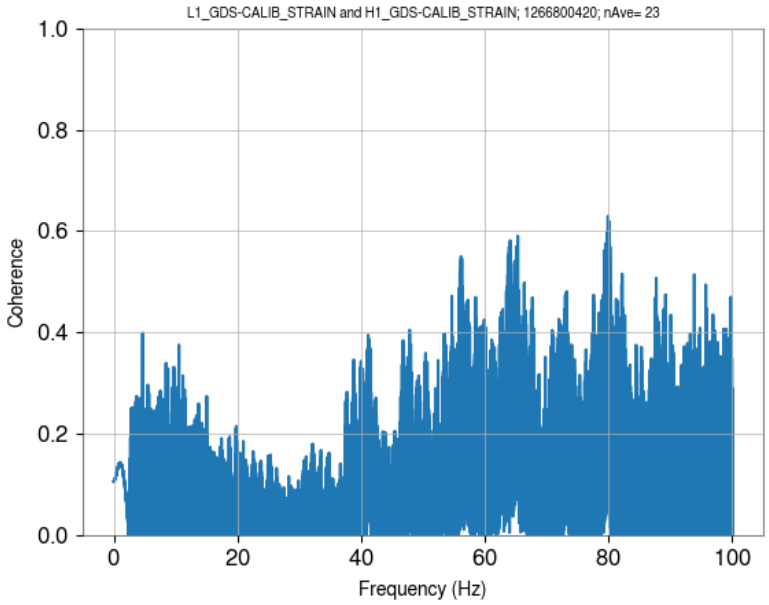

Figure 16: Sample Hanford-Livingston daily coherence data; Feb. 27, 2020 [20]

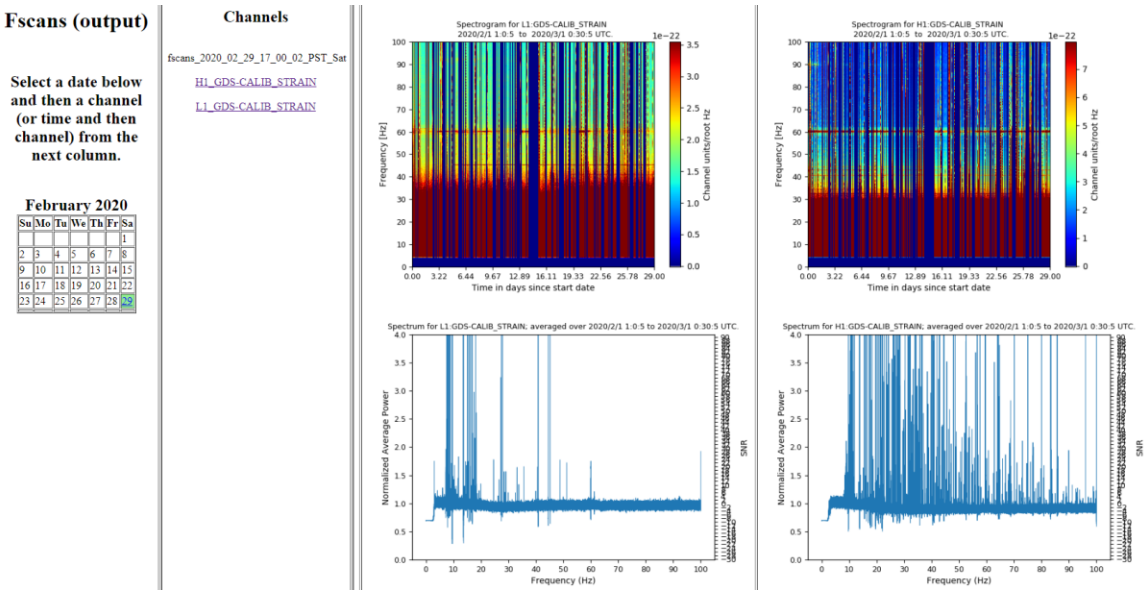

Figure 17: Sample Hanford-Livingston monthly Fscan output page [20]

Appendix G: Sample Line Density Tracking Output

```
Start working directory: /home/thomas.harris

Fscan spectra line finding script with chi2 statistics
Date target for this line counter: 2020_02
Cutoff probability criteria for high-power lines: 1e-06

Current working directory:
/home/pulsar/public_html/fscan/H1/daily/H1Fscan_coherence/H1Fscan_coherence

Current directory:
/home/pulsar/public_html/fscan/H1/daily/H1Fscan_coherence/H1Fscan_coherence/fscans_2020_02_01_17_0
0_02_PST_Sat/H1_GDS-CALIB_STRAIN
Spectrum data obtained from spec_0.00_100.00_H1_1264554020_1264640420.txt.
Degrees of freedom = 2 * number of SFTs = 2 * number of averages : 86
Normalized threshold power = 1.8985771655
Number of high power lines: 1118

Current directory:
/home/pulsar/public_html/fscan/H1/daily/H1Fscan_coherence/H1Fscan_coherence/fscans_2020_02_02_17_0
0_02_PST_Sun/H1_GDS-CALIB_STRAIN
Spectrum data obtained from spec_0.00_100.00_H1_1264640420_1264726820.txt.
Degrees of freedom = 2 * number of SFTs = 2 * number of averages : 94
Normalized threshold power = 1.85204839777
Number of high power lines: 1381

Current directory:
/home/pulsar/public_html/fscan/H1/daily/H1Fscan_coherence/H1Fscan_coherence/fscans_2020_02_03_17_0
0_02_PST_Mon/H1_GDS-CALIB_STRAIN
Spectrum data obtained from spec_0.00_100.00_H1_1264726820_1264813220.txt.
Degrees of freedom = 2 * number of SFTs = 2 * number of averages : 86
Normalized threshold power = 1.8985771655
Number of high power lines: 1269

[ FILE BREAK INSERTED ]

Current directory:
/home/pulsar/public_html/fscan/H1/daily/H1Fscan_coherence/H1Fscan_coherence/fscans_2020_02_28_17_0
0_02_PST_Fri/H1_GDS-CALIB_STRAIN
Spectrum data obtained from spec_0.00_100.00_H1_1266886820_1266973220.txt.
Degrees of freedom = 2 * number of SFTs = 2 * number of averages : 66
Normalized threshold power = 2.05475058954
Number of high power lines: 1012

Current directory:
/home/pulsar/public_html/fscan/H1/daily/H1Fscan_coherence/H1Fscan_coherence/fscans_2020_02_29_17_0
0_02_PST_Sat/H1_GDS-CALIB_STRAIN
Spectrum data obtained from spec_0.00_100.00_H1_1266973220_1267059620.txt.
Degrees of freedom = 2 * number of SFTs = 2 * number of averages : 96
Normalized threshold power = 1.84143304522
Number of high power lines: 1089


Number of days with available data: 29
Average number of daily high-power lines in H1_GDS-CALIB_STRAIN during 2020_02: 1130.24137931
Standard deviation in number of daily high-power lines: 147.810376

High line density days during 2020_02:
fscans_2020_02_09_17_00_02_PST_Sun, number of high-power lines = 1492

End working directory:
/home/pulsar/public_html/fscan/H1/daily/H1Fscan_coherence/H1Fscan_coherence/fscans_2020_02_29_17_0
0_02_PST_Sat/H1_GDS-CALIB_STRAIN
```

Bibliography

[1] K. Riles, "Gravitational Waves: Sources, Detectors and Searches," *Progress in Particle and Nuclear Physics,* vol. 68, 2013.

[2] W. R. Johnston, "Gravitational wave-related images," 1 April 2005. [Online]. Available: http://www.johnstonsarchive.net/relativity/pictures.html.

[3] K. S. Thorne, "Gravitational Radiation -- A New Window Onto the Universe," *Reviews in Modern Astronomy,* vol. 10, 1997.

[4] LIGO Scientific Collaboration and Virgo Collaboration, "Observation of Gravitational Waves from a Binary Black Hole Merger," *Phys. Rev. Lett.,* vol. 116, 2016.

[5] LIGO Scientific Collaboration and Virgo Collaboration, "GWTC-1: A Gravitational-Wave Transient Catalog of Compact Binary Mergers Observed by LIGO and Virgo during the First and Second Observing Runs," *Phys. Rev. X,* vol. 9, 2019.

[6] LIGO Scientific Collaboration and Virgo Collaboration, "GraceDB — Gravitational-Wave Candidate Event Database," [Online]. Available: https://gracedb.ligo.org/latest/. [Accessed 12 May 2020].

[7] S. Ghonge, K. Jani, LIGO, VIRGO and G. Tech, "O1/O2 Catalog," 2018. [Online]. Available: https://www.ligo.org/detections/O1O2catalog.php. [Accessed 8 May 2020].

[8] LIGO Scientific Collaboration, "Instrument Science White Paper," 6 November 2019. [Online]. Available: https://dcc.ligo.org/LIGO-T1900409/public.

[9] J. Kissel, "aLIGO Seismic Isolation and Suspensions Cartoon," 13 November 2017. [Online]. Available: https://dcc.ligo.org/LIGO-G1200071/public. [Accessed 8 May 2020].

[10] M. Tse, V. Roma, T. Hardwick and P. Nguyen, "PEM Channel Info," May 2019. [Online]. Available: http://pem.ligo.org/channelinfo/index.php. [Accessed 8 May 2020].

[11] J. McIver and L. Nuttall, "2019-2020 LIGO DetChar white paper draft," 2019. [Online]. Available: https://dcc.ligo.org/LIGO-T1900369.

[12] G. Mendell, "Introduction To Signal Processing," 2012. [Online]. Available: https://dcc.ligo.org/LIGO-G1200759.

[13] G. Mendell and M. Landry, "StackSlide and Hough Search SNR and Statistics," 2005. [Online]. Available: https://dcc.ligo.org/LIGO-T050003/public.

[14] LIGO Scientific Collaboration and Virgo Collaboration, "Gravitational Wave Open Science Center Plot Gallery," [Online]. Available: https://www.gw-openscience.org/plot_gallery/. [Accessed May 2020].

[15] LIGO Scientific Collaboration, "Fscan Monthly Coherence Navigation," [Online]. Available: https://ldas-jobs.ligo-wa.caltech.edu/~pulsar/fscan/H1/monthly/H1Fscan_coherence/fscanNavigation.html. [Accessed 8 May 2020].

[16] LIGO Detector Characterization Working Group, "LIGO Channel Lists," 2019. [Online]. Available: https://git.ligo.org/duncanmmacleod/ligo-channel-lists/-/tree/master/O3.

[17] G. Mendell, "Significance of the Magnitude Squared Coherence," 4 March 2013. [Online]. Available: https://dcc.ligo.org/LIGO-G1300070.

[18] E. Goetz et al., "aLIGO-lines-combs/O3," [Online]. Available: https://git.ligo.org/CW/instrumental/aLIGO-lines-combs/-/tree/master/O3.

[19] G. Mendell and T. Harris, "Fscan H1L1 Daily Coherence," [Online]. Available: https://ldas-jobs.ligo-wa.caltech.edu/~pulsar/fscan/H1/daily/H1L1Fscan_coherence/.

[20] G. Mendell and T. Harris, "Fscan H1L1 Monthly Coherence Navigation," [Online]. Available: https://ldas-jobs.ligo-wa.caltech.edu/~pulsar/fscan/H1/monthly/H1L1Fscan_coherence/fscanNavigation.html.

[21] T. Harris, "Fscan," 12 May 2020. [Online]. Available: https://ldas-jobs.ligo-wa.caltech.edu/~thomas.harris/fscan/.

[22] G. Mendell and T. Harris, "Fscan Coherence Report for O3," 8 May 2020. [Online]. Available: https://dcc.ligo.org/G2000677.

[23] E. Cuoco et al., "Enhancing Gravitational-Wave Science with Machine Learning," 2020. [Online]. Available: arXiv:2005.03745.

[24] M. Zevin et al., "Gravity Spy: Integrating Advanced LIGO Detector Characterization, Machine Learning, and Citizen Science," *Classical and Quantum Gravity,* vol. 34, 2017.