2011

DQTunePipe: a Set of Python Tools for LIGO Detector Characterization

Brooke Anne Rankins

DQTunePipe: A set of Python tools for LIGO detector characterization

By
Brooke Anne Rankins

A Thesis
Submitted to the Faculty of
University of Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Master of Science in Physics
Department of Physics and Astronomy

2011 December

ABSTRACT

When LIGO's interferometers are in operation, many auxiliary data channels monitor and record the state of the instruments and surrounding environmental conditions. Analyzing these channels allows LIGO scientists to evaluate the quality of the data collected and veto data segments of poor quality. A set of scripts were built up in an ad hoc fashion, sometimes with limited documentation, to assist in this analysis. In this thesis, we present `DQTunePipe`, a set of Python modules to replace these scripts and aid in the detector characterization of the LIGO instruments. The use of Python makes the analysis method more compatible with existing LIGO tools. `DQTunePipe` improves data quality analysis by allowing users to select specific detector characterization tasks as well as providing a maintainable framework upon which additional modules may be built. The nature of the Python `DQTunePipe` code allows the addition of new features with great simplicity. This thesis details the structure of `DQTunePipe`, serves as its documentation at the time of this writing, and outlines the procedures for incorporating new features.

GLOSSARY & ACRONYMS

analysis segment a time interval within a science segment when triggers are generated, *page 14*

DetChar Detector Characterization, *page 1*

DQ Data Quality, *page 1*

DQ Flag interval of time when the gravitational wave channel data may be of questionable quality, denotes periods of time when the auxiliary channels or the gravitational wave channel may be affected by noise transients, *page 13*

DQ window specific interval of time identified by a DQ flag, *page 13*

glitch artifact in data due to noise transients, *page 13*

Glue Grid LSC User Environment, *page 23*

hardware injections simulated gravitational wave signals, *page 16*

KW Kleinewelle, *page 15*

LIGO Laser Interferometer Gravitational-wave Observatory, *page 1*

LSC LIGO Scientific Collaboration, *page 1*

noise transient event of non-astrophysical origin, *page 13*

science segment interval of time within a science run that contain science data, *page 14*

SciMon Science Monitor, on-site scientist who monitors data quality, *page 16*

SNR Signal-To-Noise Ratio, denoted ρ in equations, *page 10*

tuning modifications to the DQ intervals following offline investigations, *page 19*

veto interval of time which may be either excluded entirely or only conditionally included in the search for gravitational waves, *page 14*

ACKNOWLEDGEMENTS

The author gratefully acknowledges the support of the United States National Science Foundation for the construction and operation of the LIGO Laboratory. The author also thanks LIGO and the LIGO Scientific Collaboration for allowing the use of LIGO data for this work. The author would also like to acknowledge the support of this work via NSF grants PHY-0757937 and PHY-1067985. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. Analysis results presented are intended as use case examples for the purposes of illustrating the functionality of the `DQTunePipe`, the subject of this thesis. This paper is assigned LIGO document number LIGO-P1100190, and is currently under review.

I would also like to extend my thanks to all those individuals who have supported me through this process. I would like to acknowledge my advisor, Marco Cavaglià, both for his advising in this work and for giving me the opportunity to participate in LIGO's Education and Public Outreach team. I have learned a great deal from my involvement with that group. I would also like to acknowledge my defense committee members, Emanuele Berti and particularly Lucien Cremaldi, who brought me back to the University of Mississippi to study physics in the first place. Finally, I would like to thank my family and friends, particularly Will, Jimmy and Ginger, and Kathy and Jim, who have shared with me their homes, patience and experience, for which I am forever grateful.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Gravity may be interpreted in terms of the geometry of space-time. As masses curve the space-time around themselves, their gravitational interactions with other masses become apparent. The acceleration of matter produces fluctuations or gravitational waves of the space-time the matter curves. Astrophysical objects offer the most obvious source for gravitational wave detection. Gravitational waves are the primary focus of study for the scientists involved with the Laser Interferometer Gravitational-wave Observatory (LIGO).

LIGO detectors are built for the purpose of directly detecting gravitational waves and using those detections to develop gravitational wave astronomy. More than 800 researchers work together in the LIGO Scientific Collaboration (LSC) to operate LIGO and analyze the data the instruments collect.

In this thesis, Chapter 1 presents a short introduction to gravitational waves and LIGO. Chapter 2 discusses the noise that interferes with gravitational wave detection at LIGO and presents some of the methods employed by LSC scientists to overcome this noise as is relevant to this thesis. Specifically, Chapter 2 describes Data Quality (DQ) flags, intervals of time identified as containing data of questionable quality for the purposes of vetoing some of this noisy data, as defined by scientists in the Detector Characterization (DetChar) team of the LSC.

The main result of this thesis is a Python program for investigating and fine-tuning DQ flags for use in the search for gravitational waves from compact binary coalescing (CBC) sources. Chapter 3 outlines the design and the execution flow of the program. Chapter 4 describes instructions for operating and modifying the program. Chapter 5 presents an example of DQ analysis using the program, and conclusions in Chapter 6 include prospects for further development.

1.1 Gravitational Waves

Gravitational waves may be regarded as fluctuations in the space-time curvature with a definite spatial and temporal pattern. In the absence of energy or matter, flat space-time may be described locally by the Minkowski metric,

$$\eta_{\mu\nu} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{1.1}$$

The space-time interval between two neighboring events in the Minkowski space-time is

$$ds^2 = \eta_{\mu\nu} dx^\mu dx^\nu, \tag{1.2}$$

where $\mu, \nu = 0, 1, 2, 3$; $x^\mu = (t, \mathbf{x})$; and the speed of light is defined in natural units as $c = 1$.

In the weak field regime, gravitational waves are described by small perturbations of the Minkowski metric

$$g_{\mu\nu} = \eta_{\mu\nu} + h_{\mu\nu}, \tag{1.3}$$

where $|h_{\mu\nu}| \ll 1$. The Einstein field equations,

$$G_{\mu\nu} = 8\pi G T_{\mu\nu}, \tag{1.4}$$

relate the Einstein tensor, $G_{\mu\nu}$, which is a function of the space-time curvature, to the stress-energy tensor of matter, $T_{\mu\nu}$ [16]. Making use of a transverse-traceless gauge for the metric, the metric perturbation $h_{\mu\nu}$ in vacuum satisfies the wave equation

$$\left(\nabla^2 - \frac{\partial}{\partial t^2}\right) h_{\mu\nu}(x) = 0. \tag{1.5}$$

The plane wave solution of Equation (1.5) is

$$h_{\mu\nu}(x) = a_{\mu\nu} e^{\imath k_\mu x^\mu}, \tag{1.6}$$

where

$$k_\mu x^\mu = -\omega t + \mathbf{k} \cdot \mathbf{x}, \tag{1.7}$$

and $a_{\mu\nu}$ defines the polarization tensor, which is traceless and orthogonal to k_μ:

$$a^j{}_j = 0, \tag{1.8}$$

$$k^j a_{ij} = 0. \tag{1.9}$$

For a plane wave propagating along the z-axis, $a_{\mu 3} = 0$, and Equations (1.8) and (1.9) imply

$$a_{\mu\nu} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & f & g & 0 \\ 0 & g & -f & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \tag{1.10}$$

where f, g are constants. Equation (1.10) implies that the gravitational wave is transverse and its amplitude can be expressed as the linear combination of two amplitudes, h-plus

(h_+) and h-cross $(h_\times)$, $a_{\mu\nu} = fh_+ + gh_\times$, where

$$h_+ = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad h_\times = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}. \tag{1.11}$$

These amplitudes describe the two orthogonal polarizations of the plane gravitational wave propagating along the z-axis.

LIGO's design is based on measuring the relative displacement of test masses due to the perturbation of the metric caused by a gravitational wave. Assuming that a plane gravitational wave propagates along the z-axis and impinges on a ring of test masses in the (x, y) plane as in Figure 1.1, Equation (1.6) becomes

$$h_{\mu\nu}(x) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & h_+ & h_\times & 0 \\ 0 & h_\times & -h_+ & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} e^{-\imath\omega t}. \tag{1.12}$$

An h_+ polarized gravitational wave will cause the separation of the test masses to oscillate with a phase difference of 180° along the x and y axes; the distance of two opposite test masses measured along the $x(y)$ direction will alternatively increase(decrease) with a frequency equal to the gravitational wave frequency. Likewise, an $h_\times$ polarized gravitational wave will "stretch" and "shrink" distances along 45° diagonals.

If a pair of test masses are located at a distance L_* apart along the x-axis in unperturbed space-time, then in a space-time perturbed by an h_+ polarized gravitational

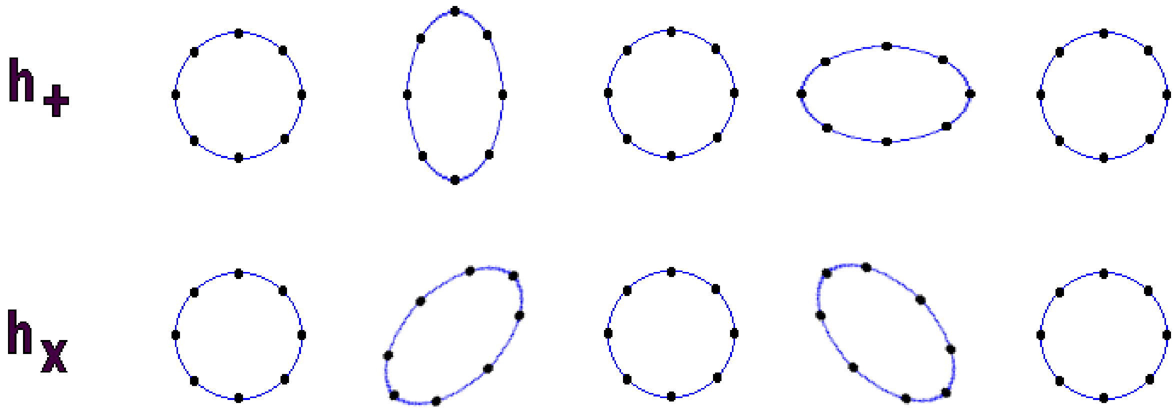


Figure 1.1: h-plus and h-cross polarizations - Illustrated by changes in distances between test masses in a plane perpendicular to the direction of travel of the propagating gravitational wave.

wave, the distance between the masses is

$$L(t) = \int_{-L_*/2}^{L_*/2} dx[1 + h_{11}(t)]^{\frac{1}{2}} \approx L_*[1 + \frac{1}{2}h_{11}(t)], \tag{1.13}$$

where the two test masses are assumed to be located at $x_1 = -\frac{L_*}{2}$ and $x_2 = \frac{L_*}{2}$, and the wavelength of the gravitational wave is much larger than L_*. The relative change in the measured distance, or strain, is

$$\frac{\Delta L(t)}{L_*} = \frac{1}{2}h_{11}(t). \tag{1.14}$$

The amplitude of the gravitational wave is therefore proportional to the gravitational wave strain, $\Delta L(t)/L_*$ [16].

1.1.1 Sources of Gravitational Waves

Gravitational waves can provide insight into cosmological phenomena. Gravitational wave sources include burst sources, such as supernovae; periodic sources, such as rotating neutron stars; compact binary coalescing sources, such as a pair of orbiting black holes; and the stochastic gravitational wave background. An in-depth discussion of these sources is beyond the scope of this writing, but a brief explanation of compact binary coalescing astrophysical systems as a gravitational wave source is relevant to this thesis.

A compact binary system is an orbiting system of two extremely dense celestial objects, such as the binary pulsar system discovered by Russell Hulse and Joseph Taylor in 1974 [17]. Over time, such a system loses energy due to gravitational radiation. The evolution of the system may be described, for the purposes of gravitational wave detection, in terms of three continuous stages. In the first stage, energy loss from gravitational wave emission causes the orbit to decay. The corresponding gravitational waveform that represents this process is known as a 'chirp'. In a chirp waveform, the gravitational wave frequency and amplitude increase as the orbit reduces in size. This stage is followed by a stage when the two spiraling bodies merge into a single object. This process is known as binary coalescence [14]. After the merger, gravitational waves are produced as the rotating object settles into a stationary state, in what is known as the "ringdown" stage. The Python program presented in this thesis is designed for use in searches for compact binary coalescing sources.

1.2 LIGO

1.2.1 Detection of Gravitational Waves

LIGO consists of twin laboratories, one located in Hanford, WA and another located in Livingston, LA. LIGO's basic instrument design is that of a Michelson-Morley interferometer. Each L-shaped interferometer has two 4km arms enhanced with Fabry-Perot resonance cavities in vacuum. The Hanford site also houses a second 2km arm

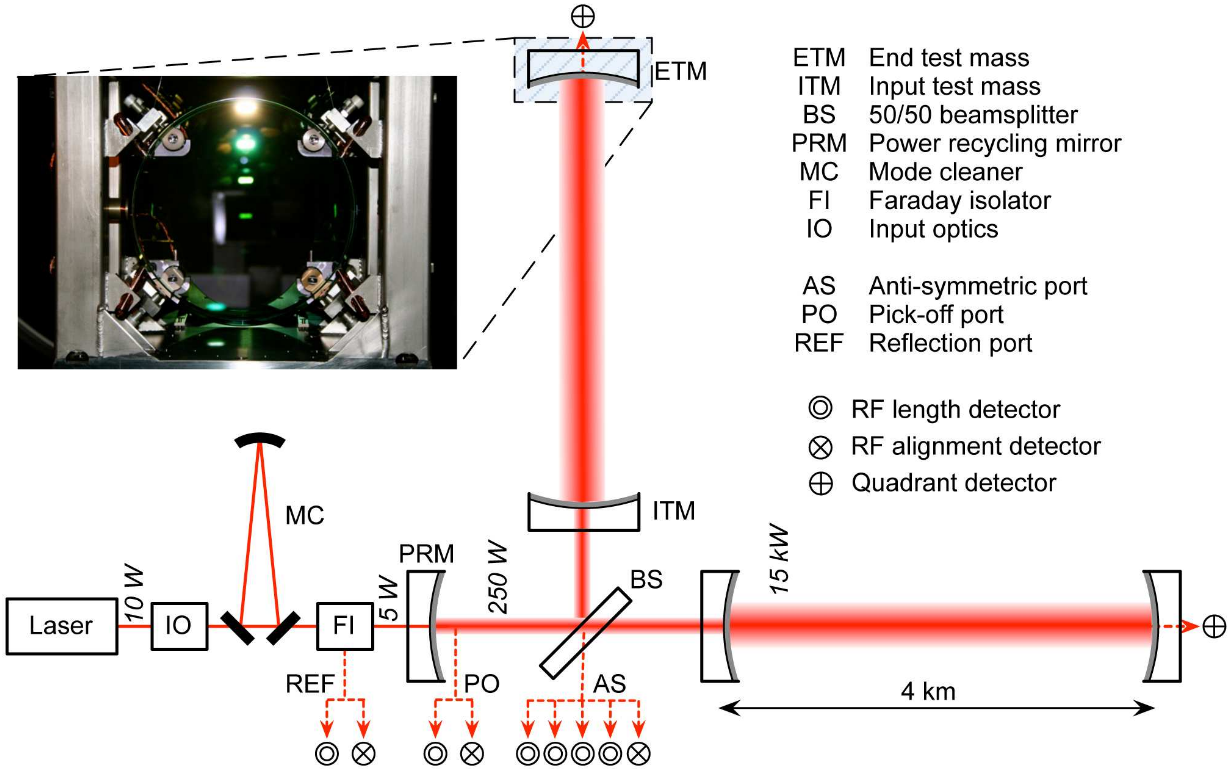


Figure 1.2: A schematic of a LIGO interferometer. - The input test mass mirrors, ITMX and ITMY along the x and y axes, form the Fabry-Perot cavity with the respective end mirrors, ETMX and ETMY.

interferometer which was not used in LIGO's most recent science run. In order to detect gravitational waves, LIGO scientists measure the differential change in length between the interferometer arms.

The basic design of the LIGO instrument is as follows. In each LIGO interferometer, a beam of stabilized infrared light is split in two via a 50% reflecting mirror (beam splitter), and each beam is directed into the cavities of the interferometer's arms, (referred to as the X and Y arms). The beams reflect off mirrors suspended at the end of each arm and are amplified in Fabry-Perot cavities before recombining. The Fabry-Perot resonators of the Michelson interferometer increase the sensitivity range [20]. The mirrors function as test masses which are displaced when a gravitational wave passes through the interferometer. Power-recycling optics are used to return stray laser beam light back into the system.

LIGO is designed such that under normal conditions the photodetector output is null.

If the mirrors' locations are altered by a gravitational wave, photodetectors measure the oscillations in the recombined beam's intensity, signaling variations in the lengths of the interferometer arms.

1.2.2 Calibration

A length sensing and feedback control system monitors and adjusts the locations of the mirrors to attain stable resonance [12]. Once achieved, the interferometer is said to be locked, and the interferometer can put into science mode, a state where data collected by the instrument is suitable for the search for gravitational waves. The Differential Arm Error (DARM_ERR) signal from the feedback loop of the control system controls the motion of the interferometer arms and serves as the gravitational wave channel. LIGO scientists reconstruct the gravitational wave strain from this error signal using the response function of the interferometer. The error signal is related to the strain (gravitational and noise) in the frequency domain $\hat{s}(f)$ by

$$\hat{s}(f) = R(f)\hat{q}(f), \tag{1.15}$$

where $R(f)$ is the response function, and $\hat{q}(f)$ is the error signal, i.e. the data from the gravitational wave channel DARM_ERR [4, 14]. The response function is dependent on the parameters of a sensing function, a digital filter function, and the actuation function. The sensing function, $C(f)$, measures the arm cavity's response to a gravitational wave and converts the residual strain to the output of DARM_ERR. It depends on the light power stored in the interferometer arms and thus changes over time. To keep the cavities locked, the digital filter, $D(f)$, converts this error signal to a control signal received by the mirrors. Single frequency sinusoidal calibration signals are continuously added to the control signals that drive the mirrors in order to measure $C(f)$. The actuation function, $A(f)$, determines the current to the electromagnets that control the positions of the mirrors and the arm cavity lengths. The residual strain ($\hat{s}_{res}$) can be thought of as what is left when the control strain ($\hat{s}_{control}$) is subtracted from the gravitational wave strain,

$$\hat{s} = \hat{s}_{res} + \hat{s}_{control}. \tag{1.16}$$

The residual strain represents the residual motion of the mirrors after the control signal has been applied to them and implies the corresponding error signal

$$\hat{q}(f) = \hat{s}_{res} C(f). \tag{1.17}$$

The control strain is

$$\hat{s}_{control} = \hat{s}_{res} G(f), \tag{1.18}$$

where $G(f)$ is the open loop gain of the interferometer. $G(f)$ is given by

$$G(f) = C(f)D(f)A(f). \tag{1.19}$$

Using the open loop gain and substituting (1.17), (1.16), and (1.18) into (1.15) gives the response function is

$$R(f) = \frac{G(f) + 1}{C(f)}. \tag{1.20}$$

It is essential to accurately determine the response function as it is vital to the reconstruction of the strain from the error signal, as seen from equation (1.15).

1.2.3 Templates and Matched Filtering

Input data used by the Python program in this thesis (to be presented in Chapter 3) is generated by comparing the gravitational wave strain to theoretically predicted waveforms from CBC sources via a matched filtering algorithm.

In the presence of a gravitational wave, the detector output time series, $s(t)$, from the gravitational wave channel is the sum of the noise, $n(t)$, and the gravitational wave signal, $h(t)$:

$$s(t) = h(t) + n(t). \tag{1.21}$$

The detector output is then digitally filtered:

$$S = H + N, \tag{1.22}$$

where H is the filtered signal and N is the filtered noise,

$$H \equiv \int_{-\infty}^{\infty} K(t)h(t)dt \quad , \quad N \equiv \int_{-\infty}^{\infty} K(t)n(t)dt, \tag{1.23}$$

respectively. The signal-to-noise ratio, or SNR (ρ), is defined by

$$\rho^2 = \frac{H^2}{< N^2 >}. \tag{1.24}$$

The filter $K(t)$ is chosen to maximize the SNR [14].

For CBC searches, the matched filtering method is favored since the theoretical gravitational waveforms are known and so one may construct an optimal filter from the Fourier transform of the signal [10, 14]. CBC searches make use of banks of template waveforms representing various physical configurations of binary systems in a range of parameter combinations. For example, CBC template waveform parameters may include the masses and spins of the binary components, the end time of the inspiral phase, the distance to the distance to the binary system, etc. [4].

1.3 Trigger Data

At each interferometer, when the signal-to-noise ratio from matched filtering output exceeds a predetermined threshold, a trigger may be produced [6]. For inspiral searches, triggers are generated such that each trigger is separated in time by the template duration or chirp time, i.e. the length of time during which a binary system will radiate gravitational waves within LIGO's sensitivity band. This duration is established for each template given a low frequency cutoff value for the detector sensitivity band. When a trigger is created, its SNR value is compared against SNR values for other triggers within the chirp time. Only the trigger within the chirp's duration with the highest SNR value is recorded, so that the set of triggers are distanced in time by at least one chirp duration.

During data analysis, the list of triggers is reduced by eliminating those triggers that do not coincide in time and waveform parameters at both detectors. The template parameters of this smaller collection of triggers are then used to generate a new template bank, and using this bank, matched filtering is again run on the data [19]. After the second matched filtering, consistency checks, such as a χ^2 test, are used to compare the data to the template to discriminate between realistic candidate events and noise [9]. Triggers that survive the consistency tests are again subjected to a coincidence test between interferometers. At various stages in the analysis pipeline, the list of triggers may be reduced further by vetoes identifying data of questionable quality, a process to be described in Chapter 2. Figure 1.3 illustrates the analysis pipeline. Among the information included in the recording of the triggers are the time of the event, the SNR, the mass parameters of the template, and the χ^2 veto parameters [2].

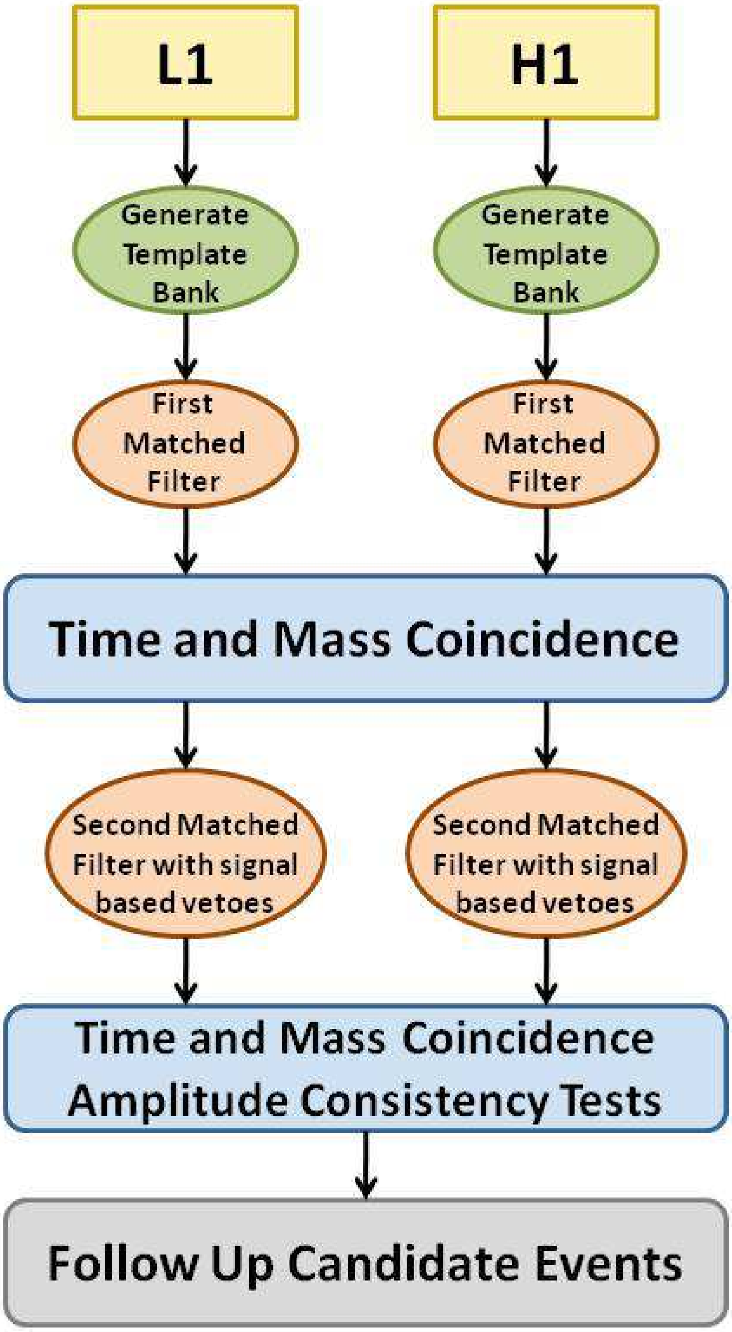


Figure 1.3: Illustration of CBC analysis pipeline - The analysis pipeline reduces the number of triggers, narrowing down possible gravitational wave candidate events. Vetoes are applied at various stages throughout the pipeline. *H1* and *L1* indicate the interferometers at Hanford, WA, and Livingston, LA, respectively.

CHAPTER 2

DATA QUALITY STUDIES

A variety of disturbances of non-astrophysical origin may contaminate the output of the gravitational wave channel. The sources of these disturbances have their origin in either the surrounding environment or the instrument itself. To combat this noise, in addition to the gravitational wave channel, LIGO also monitors hundreds of auxiliary data channels which record the state of the instruments and the local environmental conditions. The data from these channels is used by members of the DetChar group to identify the origin of noise.

2.1 Data Quality

For the purposes of this thesis, *noise transients* refer to short-lived events of non-astrophysical origin that may affect data recorded by the gravitational wave channel. Transients contribute to the background signal of the detector with noise that may potentially mask real gravitational wave signals. Instrumental or environmental transients produce *glitches* in the data stream that are recorded in the auxiliary channels and possibly the gravitational wave channel. Particularly strong noise transients may cause the interferometer to lose lock [11].

Data Quality (DQ) flags denote periods of time when auxiliary channels and/or the gravitational wave channel may be affected by noise transients. A DQ flag's name typically refers to either the likely cause of the noise transient or the auxiliary channel that records the noise transient. A DQ *window* denotes a specific interval of time identified by a DQ flag.

A science run at LIGO refers to an extended period of time when data for astrophysical analysis are collected. Science data are collected while the LIGO interferometers are in science mode. The intervals of time within a science run that contain science data are *science segments*. The subset of science segments from which triggers are generated are called *analyzed segments*.

A *veto* segment denotes an interval of time which is removed from the analysis to some degree. Vetoed data may either be excluded entirely or only conditionally included in the search for gravitational waves. DQ flags are typically used to generate vetoes [6].

Data quality flags may be established *online*, while interferometer data is being recorded, or *offline* from additional knowledge obtained after the interferometer data is collected. Online DQ flags defined from auxiliary channels are derived from monitoring either instrumental or environmental noise through the use of automated scripts.

2.1.1 Online DQ Flags

Examples of online DQ flags derived from the auxiliary channels monitoring the instrument include, among others, control channel overflows, calibration line dropouts, and light dips. *Overflows* occur when the amplitude of the feedback control signals used to control the interferometer arm lengths and mirror alignments exceed the maximum allowable amplitude that the processing channel can handle. *Calibration line dropouts* are caused by discontinuities in the calibration signal described in Section 1.2.2. *Light dips* are due to brief mirror misalignments that cause the power in the Fabry-Perot arm cavities to decrease relative to the average power over some previous time interval. [6, 8, 10].

In some cases, Data Monitoring Tool (DMT) systems are used to generate DQ flags from the auxiliary channels. For example, a DMT called LightMon examines the channels which monitor the photodetectors in the arm cavities and compares the minimum interferometer arm power value in a given second to the mean value in the previous 10 seconds. If the interferometer experiences a X percentage dip in power, LightMon creates the associated lightdip DQ flags *IFO*:DMT-LIGHTDIP_X_PERCENT (*IFO* denotes the interferometer) [15].

Environment-driven online DQ flags are based on the physical environmental monitoring (PEM) channels. The PEM channels record the output of various sensors placed at the interferometer sites. These sensors include, magnetometers to detect electromagnetic fluctuations, microphones to monitor overhead air traffic, and seismometers and accelerometers to record man-made disturbances and seismic activity.

In the case of air traffic, a DMT called PlaneMon evaluates the excess power in the microphone channels (such as PEM-BSC5_MIC). If the excess power is deemed plausible to be from a overhead airplane, PlaneMon creates the DQ flags *IFO*:DMT-AIRCRAFT_LIKELY and *IFO*:DMT-AIRCRAFT_VERY_LIKELY accordingly [13, 15].

A wavelet-based algorithm known as the Kleinewelle (KW) algorithm can also be used to establish vetoes. Kleinewelle produces single interferometer triggers for data quality purposes from the auxiliary and gravitational wave channels [11]. These *KW triggers* have some characteristics similar to the gravitational wave triggers described in Section 1.3; the KW triggers have a peak time and peak significance. The peak significance corresponds to the amplitude of the triggers. Coincidence of the KW triggers produced from the auxiliary channels and those produced from the gravitational wave channel allows the DetChar team to establish auxiliary channel vetoes [10, 18]. The Used Percentage Veto (UPV) set of flags are derived from KW triggers. The UPV tool examines the percentage of KW triggers in the auxiliary channels coincident with KW triggers in the gravitational wave channel above a specified KW significance threshold. If glitches in the gravitational wave channel are coincident with glitches in an auxiliary channel more than 50% of the time, a UPV flag for that channel is produced [7, 18].

2.1.2 Offline DQ Flags

Offline flags are a necessity for unforeseen circumstances, i.e. when automated systems are not in place to detect a particular noise source. These flags may be created as a result of ongoing DetChar studies after data collection or by Science Monitors (SciMon) who are on-site to monitor the data quality. DQ flags created by SciMons are assigned the prefix *SCI* in their nomenclature, and those created as a result of DetChar studies are designated by *DCH*.

For example, during the most recent science run denoted as S6, DetChar studies determined that some noise transients in the gravitational wave channel at the Hanford (H1) site coincided with the hourly computer back-ups at that observatory. A DQ flag, H1:DCH-AUTOBURT_GLITCH_WIDE, was created to generate time stamps from the back-up times suspected of producing the transients. Additionally, it was found that large CPU loads generated these glitches, and so associated DQ flags (H1:DCH-CPU_ASC312_SOS246, H1:DCH-CPU_ASC316_SOS251, H1:DCH-CPU_ASC316, and H1:DCH-CPU_SOS251) were created. This observed problem was corrected by limiting the back-ups at H1 to times when the interferometer was not in science mode [8, 21].

2.1.3 Vetoes

The significance of a noise transient's effect on the gravitational wave channel data varies. Therefore, the DetChar group categorizes vetoes established from DQ flags based on the severity of the transient. As of S6, categories are numbered 1 through 5. Category 1 vetoes are the most severe, and, excluding the specific category assigned to hardware injections (to be described later), DQs of each sequential category (higher numbers) are typically considered to produce weaker effects than the previous category.

Categorization allows veto choices in gravitational wave searches to be fine-tuned, since a purpose of vetoes is to reduce the rate of false alarms in gravitational wave detection searches. LIGO analysts may apply vetoes at all category levels, or they may choose to only apply vetoes selectively depending on the analysis.

Category 1 vetoes describe intervals of time when the interferometer is not operating within its configuration parameters. Any data collected during these periods is never analyzed. Most category 1 vetoes are excluded by design because the interferometer is not in science mode. (Incidentally, a DQ flag, *IFO*:DMT-OUT_OF_LOCK, has been created to distinguish these periods). However, some category 1 vetoes are generated from DQ flags defined during science time. These time intervals, such as periods when the calibration is bad because the loop gain is outside physical range (*IFO*:DMT-BAD_GAMMA) [15] are excluded in gravitational wave searches.

Category 2 vetoes identify times with well-understood coupling between the noise transients and the gravitational wave channel. The overflow DQ flags described in Section 2.1.1, such as *IFO*:DMT-ASC_OVERFLOW or *IFO*:DMT-LSC_OVERFLOW, are examples of DQ Flags that are used to define category 2 vetoes in the CBC searches [5].

A few DQ flags identify periods when hardware injections, i.e. simulated gravitational wave signals, are injected into the interferometer [6]. Hardware injection vetoes are classified depending on the type of injection and the search. For example, in the CBC inspiral searches, time intervals corresponding to DQ flags identifying inspiral hardware injections are assigned category 3. Time intervals corresponding to unmodeled hardware injections (BURST injections) are vetoed at category 2 [1].

Some DQ flags identify times when the gravitational wave channel shows an apparent correlation with some auxiliary channels even though the coupling mechanism is not well-understood. These DQ flags are used to define category 4 vetoes in S6 CBC searches.

Category 5 vetoes identify periods when the gravitational wave data may be marginally affected by noise transients. The AIRCRAFT flags generated by PlaneMon are an example of DQ flags used as category 5 vetoes in S6. These flags are typically only used in the follow-up evaluation of gravitational wave candidates [4, 10].

2.1.4 Data Quality Flag Metrics

The DQ flag metrics, or *figures of merit*, are a set of quantities that are used to evaluate the effectiveness of an individual data quality flag, establish its merit as a veto, and determine its applicable veto category.

The *deadtime*, D, is the percentage of cumulative (science) time flagged by a given DQ flag,

$$D = \frac{T_V}{T} * 100\%, \tag{2.1}$$

where T_V is sum of all flagged time, and T is the total science mode time.

Under the assumption that all triggers are of non-astrophysical origin, the effectiveness of a DQ flag as a veto in removing triggers above a specific SNR threshold is measured with the *efficiency*, E. The efficiency of a flag is the percentage of flagged triggers above a given SNR threshold, ρ_0:

$$E(\rho_0) = \frac{N_t(\rho_0)}{N_T(\rho_0)} * 100\%, \tag{2.2}$$

where N_t is the number of triggers with $\rho > \rho_0$ within the DQ windows, and N_T is the total number of triggers with $\rho > \rho_0$ in the science time.

One useful quantity in evaluating a DQ flag's effectiveness is the ratio of the efficiency to the deadtime. A ratio > 1 indicates that more triggers are flagged than would be expected by a random selection. This ratio may also be applied to evaluate the safety of a veto. A veto is considered safe if it does not falsely dismiss a statistically significant percentage of signal injections. When hardware injections are correlated with other DQ flag intervals, then the DQ's "efficiency" in removing hardware injections, E_{hi}, is compared to the flag's deadtime. If $\frac{E_{hi}}{D} \gg 1$, then the veto is considered unsafe and is rejected [10].

The *use percentage* (U) of a DQ flag is the percentage of flagged intervals that contain at least one trigger above the SNR threshold, ρ_0:

$$U(\rho_0) = \frac{N_{wt}(\rho_0)}{N_w} * 100\%, \tag{2.3}$$

where $N_{wt}(\rho_0)$ is the number of DQ windows containing at least one trigger with $\rho > \rho_0$, and N_w is the total number of windows for the given flag.

In S6, a further metric based on a chi squared χ^2 test, was proposed:

$$\chi^2(\rho_0) = \sum_{k=1}^{N_w} \frac{(n_k(\rho_0) - T_k\langle n_t(\rho_0)\rangle)^2}{T_k\langle n_t(\rho_0)\rangle}, \tag{2.4}$$

where T_k is the duration of the k^{th} window, n_k is the number of triggers above the SNR threshold ρ_0 for that window, and $\langle n_t\rangle$ is the average trigger rate above ρ_0. The χ^2 statistics measure the correlation between triggers and DQ flag windows; the larger the value of χ^2, the more effective the veto.

Table 2.1 summarizes how these metrics are used to classify DQ flags and eventually define veto categories. Category 1 vetoes and hardware injection vetoes, being intrinsically defined, are not assigned based on these metrics.

2.1.5 Category Tuning

Over time, a particular DQ flag may require *tuning*. Often tuning involves padding, i.e. adding (or subtracting) extra time to the durations of each window. This need is due to the fact that DQ flags associated with auxiliary channels identify intervals as indicated by those channels, but the noise sources that those channels identify may produce triggers not fully coincident with the DQ flag windows. Triggers occurring near to the flagged times are closely examined, and padding durations are determined as necessary. Identical padding is added to all windows of a given DQ flag.

In instances of padding, a single DQ flag may be used to produce vetoes of multiple categories. A different category is assigned for each of the DQ flag's padding scenarios. This usually means that the DQ flag will be used to generate two different vetoes. For instance (in S6), the *IFO*:DMT-ASC_OVERFLOW flag is used to produce category 2 vetoes during the windows identified by that flag. A category 4 veto is produced from this

Category	Description of vetoes	DQ flag Metric Parameters
1	Vetoes identifying times when the interferometers were not functioning within the operational design parameters.	
2	Vetoes produced from DQ flags identifying times when the coupling mechanism between the auxiliary channel and the gravitational wave channel is well understood.	These DQ flags often have low deadtimes, high efficiency at high SNR thresholds, high efficiency to deadtime ratios, and large χ^2 values.
3	In CBC searches, vetoes that describe CBC hardware injections, artificial simulations of gravitational wave data designed as a check to measure LIGO's ability to identify gravitational waves.	
4	Vetoes whose coupling may be only partially understood.	DQ flags producing vetoes of this type tend to have higher deadtimes, smaller efficiency to deadtime ratios, and lower use percentages.
5	Vetoes with low statistical significance.	These DQ flags typically have low efficiencies, small χ^2 values, high deadtimes, and consequently high use percentages.

Table 2.1: Summary of veto categories using CBC Data Quality veto category conventions [10]

flag by adding 8 seconds preceding and following the original windows, since the noise source typically produces additional triggers with lower SNR during the 8 seconds prior and after the flagged interval.

In other cases of tuning, as the issues that created the DQ flag are resolved, the category assignment may be downgraded to a weaker category or removed as a veto altogether, to reflect the improved circumstances. For instance, in S6c (c denotes the third science data collecting period in S6), LIGO scientists investigating data quality of the H1 interferometer removed the use of vetoes produced from the H1:DMT-LIGHTDIP_6_PERCENT flag for lowmass searches. Vetoes produced from the H1:DMT-LIGHTDIP_9_PERCENT flag were adopted. DetChar analysis determined that the interferometer needed to experience at least a 9% dip in power for the data quality in lowmass searches to be affected.

CHAPTER 3

DESIGN OF THE DATA QUALITY TUNE PIPE

Modifications to the LIGO detector or search techniques during or between LIGO's science runs mean that vetoes need to be revised on a continuous basis. Existing DQ flags may need to be recategorized or have their paddings adjusted. New DQ flags require categorization and padding assignments. New metrics to determine category assignment may be considered. Until S6, a number of `MATLAB` scripts were used to calculate the flags' figures of merit and other qualities useful for `DQ` categorization of the CBC searches. However, these `MATLAB` scripts would need, sometimes extensive, maintenance to accommodate these revisions. Because the scripts were designed in an ad hoc fashion and lacked proper documentation, this maintenance could be very difficult and prone to error.

The Python program `DQTunePipe` overcomes these shortcomings and improves the characterization of DQ flags for the CBC searches. In addition to calculating the necessary metric statistics described in Chapter 2, the DQTunePipe also allows users to:

1.) Easily run isolated portions of the program;

2.) Rerun the program on existing data without repeating the retrieval of all raw data;

3.) Allow user-specified configuration parameters;

4.) Incorporate new tasks into the program code in a straightforward manner.

In order to satisfy these requirements, `DQTunePipe`'s design differs from the older

`MATLAB` scripts in two major ways. First, the DQTunePipe accepts extensive configuration options regarding the data to be processed and the specification of the tasks to be run. Representing these configuration options in a file with a simple syntax allows the user to create reproducible data that may be used to tune DQ flags precisely. Second, DQTunePipe provides an abstraction layer, in the form of the `DataSet` class, over the raw data files. This data layer frees future developers from worry concerning the details of the raw data files. The remainder of this chapter discusses how other elements of the design of `DQTunePipe` contribute to achieving the goals outlined above. To provide context for understanding the design of the program, we also present an overview of the execution flow of the program.

Three main stages are executed sequentially during a `DQTunePipe` run. During the initial stage the environmental configuration is processed and raw data is acquired. The second stage uses the configuration and raw data from the first stage to build intermediate data input. The final stage instantiates the `DataSet` object and executes the defined *tasks*.

3.1 Configuration of DQTunePipe

In order to give the user full control over the parameters of each run, `DQTunePipe` accepts a large number of configuration parameters via both command line options and a configuration file. As the first step of execution, `DQTunePipe` parses these options, giving precedence to command line arguments over configuration file specifications if necessary. The result of this step is in an execution environment that is available throughout the rest of the program. This execution environment includes the `properties` object which contains all of the configuration parameters, and the `log` file used record `DQTunePipe`'s progress.

`DQTunePipe`'s *`initialize.py`* module establishes the `log` file as well as the `properties` object. The function `getConfiguration()` (found in *`configuration.py`*) sets the values for `properties.`*`attribute`*, using either default values defined by the `Properties` class, or values assigned by the user. The `Properties` class serves as a central reference to all the properties available to `DQTunePipe`. Examples of these properties include gps start time

(INITIAL_START), gps end time (INITIAL_END), which interferometer data to use (IFOS), snr threshold values (THRESHOLDS), etc.

3.2 Raw Data

Following the environment set-up, in its default configuration, the DQTunePipe control structure's next operation is to retrieve raw data. DQTunePipe uses LIGO's S6 Segment Database Tools available in the Grid LSC User Environment (Glue), specifically ***ligolw_dq_query***, [3] to generate a list of all DQ flags defined at the gps end time specified by the user. Using only unique DQ names from that list, DQTunePipe then queries the ligo segment database via ***ligolw_segment_query*** to obtain an XML file for each DQ in the list over the gps start and end times specified by the user. These DQ XML files contain the union of the segments (*windows*) of the latest version of the DQ flag over the specified period, and are placed in a separate directory, identified by DQ_XML_DIRECTORY. If any existing files are present in DQ_XML_DIRECTORY, then DQTunePipe checks to determine if DQ_LIST_FILE (the output DQTunePipe derived from ***ligolw_dq_query***) is present. If DQ_LIST_FILE is not found, then the user has supplied his or her own DQ_XML_DIRECTORY by setting a value in the configuration file, and it is assumed that all appropriate DQ data has been supplied. If DQ_LIST_FILE is present, then DQTunePipe verifies that every relevant DQ XML identified by DQ_LIST_FILE is present in DQ_XML_DIRECTORY, and queries the segment database for any missing DQ XML files.

To obtain the triggers, DQTunePipe simply copies the XML trigger files that are located in TRIGGER_SOURCE_DIRECTORIES, specifically those that match the filename formats identified by CLUSTERED_FORMAT and UNCLUSTERED_FORMAT. Finally, if necessary, DQTunePipe gets a copy of the *vetodefiner file* (VETODEFINERFILE_FULLPATH) provided by the user. The *vetodefiner file* contains the prior established veto category and padding definitions; it is either retrieved from www.lsc-group.phys.uwm.edu via wget, or copied directly from the path indicated by the user. These raw data files are stored in the directory structure imposed by DQTunePipe, illustrated in Figure 3.1.

3.3 Intermediate Data

After retrieving raw data, the next process in DQTunePipe is to generate intermediate data. DQTunePipe extracts the trigger data that the program requires from the raw trigger XML files and stores that data in more manageable text files. The trigger text files consists of two columns containing the trigger (chirp) time and the snr value from the template that generated the trigger.

Two sets of trigger data are necessary, *clustered* and *unclustered triggers*. (In CBC searches, triggers may be grouped over predetermined time intervals; the trigger with the maximum snr value is selected as the representative trigger for each period.) DQTunePipe uses the start time, end time, and filename formats specified by the user to identify the appropriate triggers and create the *unclustered* and *clustered* trigger text files.

In addition to the trigger text files, a text file containing the analyzed segments is generated from the unclustered trigger XML files. This text file contains four columns. The first column numbers each of the segments, and the last three columns identify the starting time, ending time, and duration of each segment, respectively.

3.4 The DataSet

With these initial set-up operations (environment configured, raw data retrieved, and intermediate data generated) complete, the main program is ready to execute. This process begins by instantiating a DataSet object, which is fundamental in enabling DQTunePipe to satisfy the requirements described at the beginning of this chapter, particularly requirement 4.

The DataSet represents the information necessary to run DQTunePipe, so new tasks will use DataSet without the need to create extraneous temporary files to organize data, which was an issue with the MATLAB scripts. The DataSet is built from the raw DQ XML files and

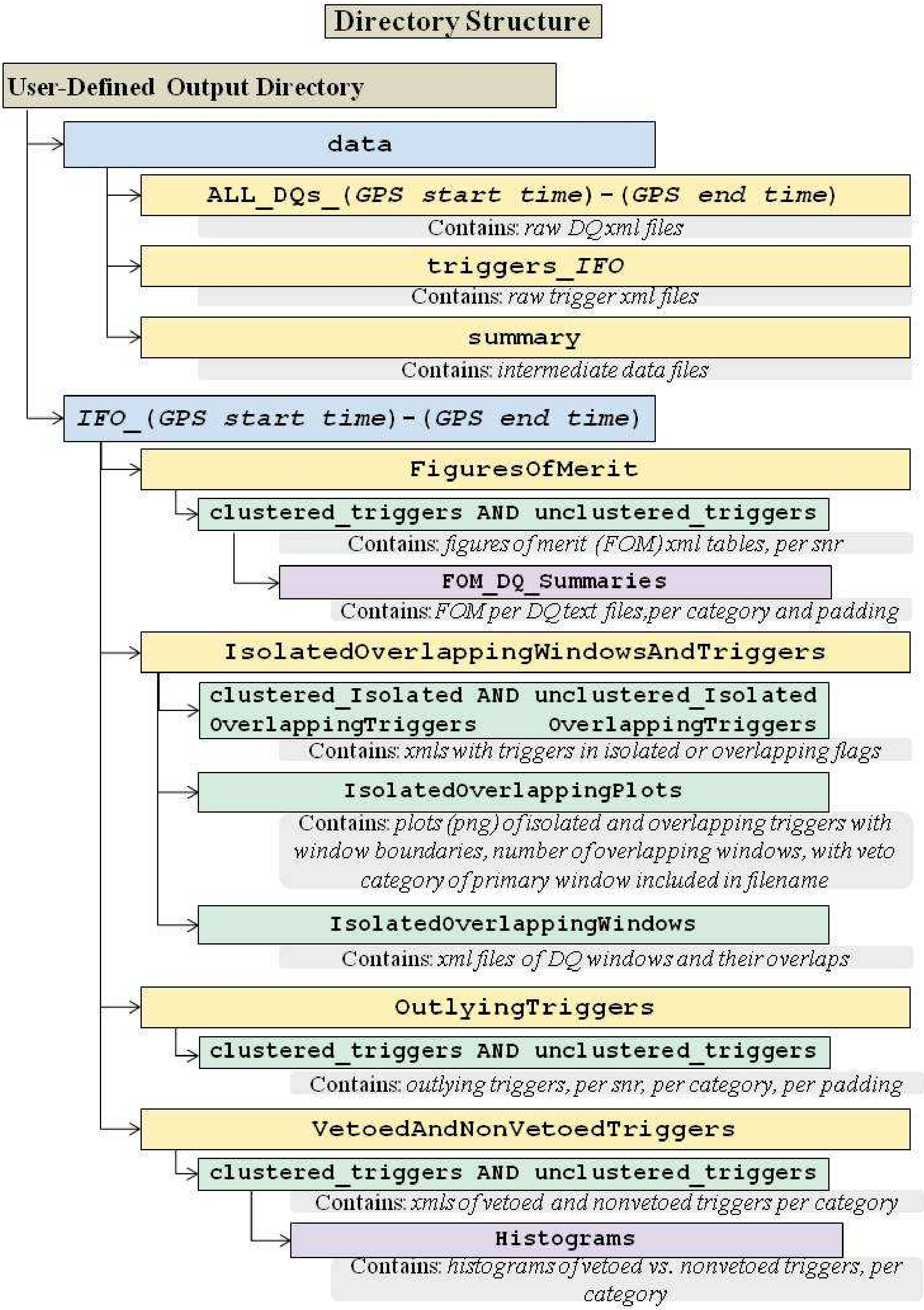


Figure 3.1: Directory hierarchy - Directory names with descriptions of files and other subdirectory names contained within. `IFO` indicates interferometer (*H1, L1*); `GPS Start` and `GPS End` indicate the starting and ending gps time the user has specified.

the intermediate trigger text files, and other variables defined from the configuration parameters. It is constructed by a call to `createDataSet(ifo)` where `ifo` refers to the interferometer whose data is to be used. Appendix C.1.1 serves as documentation for this class.

`createDataSet(ifo)` first creates an instance of a `DataSet` for the interferometer `ifo`, the user-specified science flag name for that interferometer, and the user's selected start and end gps times (see Section 4). `createDataSet` then proceeds to add the DQ information, specifically the DQ flag names and associated windows to the `DataSet` object, using the raw DQ XML files. Next, if the padded data is to be evaluated, `createDataSet` adds the padding information to the `DataSet`, using the information contained in the user-provided *vetodefiner file*. Lastly, `createDataSet` adds the trigger information to the `DataSet`, retrieving each trigger's *time* and *snr* from the clustered and unclustered trigger text files created during the intermediate data stage.

3.5 Tasks

With the `DataSet` established, the program is ready to execute its `tasks`. Since the tasks rely solely on the `DataSet`, and not directly on the raw or intermediate data files, this means that changes to the raw data's format do not require modifications to the `tasks`. Each `task` of `DQTunePipe` is represented by an object(s). The following sections outline the object classes associated with each task.

3.5.1 Figures of Merit

The `FiguresOfMerit` class produces the output XML table containing generalized overview of the veto metrics described in Section 2.1.4. An output XML file is created for every snr threshold upon which the metric quantities are calculated, and each has a listing for every `DQ` flag object in the `dataSet`. These XML tables are designed to be viewed with a ligo lightweight stylesheet, identified as `LIGO_LW_XSL`, which is included with the source

identifierTable

1: ifo	2: startTime	3: endTime	4: snr	5: triggers-Clustered	6: FileGeneratedAt
H1	968457615	968803215	6	13675	Fri Nov 18 04:46:08 2011

DQDataTable-PD

1: ifo	2: DQname	3: category	4: #of windows	5: SNR	6: deadTime (percentage)	7: efficiency (percentage)	8: usePercent	9: efficiency/ deadTime	10: chiSquare dValue
H1	BCV-KW_H1_LSC_PRC_CTRL_H1_ASC_WFS4_IP_1P66E_NEG06	category=4; multiple paddings may be applicable	20	6	0.022	0.265	65	11.887	214.843
H1	BCV-KW_H1_LSC_PRC_CTRL_H1_ASC_WFS4_IY_1P66E_NEG06	category=4; multiple paddings may be applicable	27	6	0.029	0.388	70.37	13.545	327.934
H1	BCV-KW_H1_LSC_PRC_CTRL_H1_LINEAR_1P66E_NEG06	category=4; multiple paddings may be applicable	41	6	0.049	0.633	75.61	12.91	484.393
H1	DCH-ALL_SAFE_UPV	category=4; Lpad=0.0; Rpad=0.0; applicable=(Window[961545615.0; 971654415.0; 2])	1288	6	2.36	11.678	42.547	4.948	5322.24
H1	DCH-BADGAMMA_GT8SEC	category=1; Lpad=0.0; Rpad=0.0; applicable=(Window[937473702.0; 968803215.0; 1])	12	6	0.014	0	0	0	0.666
H1	DCH-INJECTION_INSPIRAL_BLIND	category=3; Lpad=0.0; Rpad=0.0; applicable=(Window[961545615.0; 968803215.0; 1])	1	6	0.05	0.061	100	1.225	0.124
H1	DCH-INJECTION_STOCHASTIC	category=1; Lpad=0.0; Rpad=0.0; applicable=(Window[961545543.0; 968803215.0; 3])	0	6	0	0	nan	nan	0
H1	DCH-LDAS_C02_NOT_CALIBRATED	category=1; Lpad=0.0; Rpad=0.0; applicable=(Window[961545543.0; 968803215.0; 2])	23	6	0.14	0	0	0	6.871
H1	DMT-INJECTION_BURST	category=2; Lpad=16.0; Rpad=64.0; applicable=(Window[937473702.0; 968803215.0; 1])	7	6	0.46	0.572	71.429	1.231	28.524
H1	DMT-INJECTION_INSPIRAL	category=3; Lpad=0.0; Rpad=0.0; applicable=(Window[937473702.0; 968803215.0; 1])	5	6	0.28	0.306	60	1.075	32.791
H1	DMT-ISI_OVERFLOW	category=2; Lpad=0.0; Rpad=0.0; applicable=(Window[937473702.0; 968803215.0; 1])	6	6	0.0029	0	0	0	0.143
H1	DMT-ISI_OVERFLOW	category=4; Lpad=8.0; Rpad=8.0; applicable=(Window[937473702.0; 968803215.0; 1])	23	6	0.091	0	0	0	4.446
H1	DMT-LIGHTDIP_9_PERCENT	category=4; Lpad=0.0; Rpad=0.0; applicable=(Window[937473702.0; 968803215.0; 1])	13	6	0.0068	0	0	0	0.333
H1	DMT-LSC_OVERFLOW	category=2; Lpad=0.0; Rpad=0.0; applicable=(Window[937473702.0; 968803215.0; 1])	10	6	0.0058	0	0	0	0.285
H1	DMT-LSC_OVERFLOW	category=4; Lpad=8.0; Rpad=8.0; applicable=(Window[937473702.0; 968803215.0; 1])	22	6	0.089	0	0	0	4.351

Figure 3.2: Example of Figure of Merit Table - an excerpt of calculated metrics for padded data, as displayed in browser with ligo lightweight stylesheet. Unpadded data is similarly displayed but lacks the category and padding information column. The category column indicates the padding specified by the vetodefiner file, and links to the vetodefiner file if either multiple paddings, or no paddings from the vetodefiner file are applicable.

code of `DQTunePipe`. The stylesheet has been modified from the standard *ligolw.xsl* in use with other LIGO lightweight XML files to support displaying links, and is copied into the directories housing these XML files by the `FiguresOfMerit` class.

The `FiguresOfMerit` class relies on the class `Metric`, which calculates the *efficiency*, *efficiency/deadtime*, *use percentage*, and χ^2 outlined in in Section 2.1.4. The `Metric` also calculates and stores additional useful information about the veto metrics, which is written

```
Output for IFO: L1 and DQ Flag: DCH-OMC_INPUT_ANGULAR_MED
Time analyzed: 254148.0sec
Number of inspiral clusters: 18395
Mean time between clusters: 13.816
Median snr of inspiral clusters: 5.938
Maximum snr of inspiral clusters: 6574.139
Veto window buffer:  cat: 4, (0.0, 0.0) sec
 applicable over (961545615.0, 971654415.0, version:1)

Number of veto windows: 11293
Median length of veto windows: 2.0
Deadtime: 9.305%
Max chiSquared Value (SNR, chiSquared):  at SNR > (1000, 207918.296)

SNR >  vetoed/total  Efficiency  Eff/deadTime Used Windows/Total UsedPercent
    6     1436/8136     17.65 %          1.90         1378/11293     12.20 %
    8      120/544      22.06 %          2.37          116/11293      1.03 %
   12       51/286      17.83 %          1.92           51/11293      0.45 %
   20       36/187      19.25 %          2.07           36/11293      0.32 %
   50        27/85      31.76 %          3.41           27/11293      0.24 %
  100        21/51      41.18 %          4.43           21/11293      0.19 %
  200        17/24      70.83 %          7.61           17/11293      0.15 %
  500        15/15     100.00 %         10.75           15/11293      0.13 %
 1000        11/11     100.00 %         10.75           11/11293      0.10 %
```

Figure 3.3: Example of Figures of Merit Summary for the L1:DCH-OMC_INPUT_ANGULAR_MED flag

to a text file for each DQ flag. These text files contain a detailed summary of information used in calculating the metrics, including the total number of DQ windows used and the degrees of freedom in determining the χ^2 value. Additionally, this text file contains the metric quantities for all `DQ`s at every snr threshold calculated (`THRESHOLDS`), to allow the user to evaluate the metric quantities from different snr thresholds for the same DQ flag.

A configuration option (`--fom`) allows users to choose whether to execute this task explicitly. The classes for calculating the veto metrics and writing them have been separated from each other. Future LIGO users may want to change the format of this output, or perhaps put it into a database as the categorization process becomes more automated. Establishing the `Metric` object separately means the fundamental metric calculations can be executed regardless of the output format.

3.5.2 Isolated and Overlapping Windows and Triggers

Often, multiple auxiliary channels may create DQ flags during the same interval of time. When a glitch occurs during a DQ flag window whose duration overlaps with a window of another DQ flag, it may indicate a connection between the two DQ flags, particularly if the same two flags consistently have overlapping windows. It is therefore of interest to LIGO detector analysts to determine whether or not a triggers, and by extension, `DQ` windows, are *isolated* or *overlapping*.

`DQTunePipe` has a task to identify isolated and overlapping DQ windows, with specific category rules imposed. For the CBC inspiral search, (where hardware injections are assigned to category 3 [1]), `DQTunePipe` defines a DQ flag window (*self*) as *isolated* if one of the following conditions is met:

- There are no *other* DQ flag windows which overlap in time with it (*self* window).
- The *self* window is of category 1 and all *other* DQ flag windows with which it overlaps in time are not category 1.
- All *other* DQ flag windows with which it overlaps in time are of a category value greater than its (*self*) category, and the *other* category value is not assigned to an injection of the search.

The `OverlappingWindows` class of `DQTunePipe` defines the object associated with the task to determine which windows are isolated and which are overlapping. The `makeOverlappingWindows` method applies the category rules to determine the overlapping windows.

For clarification, Table 3.5.2 outlines the overlapping category rules. For example, if category 4 applies to *self* window, and category 2 applies to the *other* window which overlaps with *self*, then the *self* window is considered an *overlapping* window. Likewise, if

Category applicable to window of		***self* window** is considered	
self	*other*	isolated	overlapping
1	1		X
1	2	X	
1	3	X	
1	4	X	
1	5	X	
2	1		X
2	2		X
2	3		X
2	4	X	
2	5	X	
3	1		X
3	2		X
3	3		X
3	4	X	
3	5	X	
4	1		X
4	2		X
4	3		X
4	4		X
4	5	X	
5	1		X
5	2		X
5	3		X
5	4		X
5	5		X

Table 3.1: Category rules of determining whether a DQ window is considered *isolated* or *overlapping* for CBC searches - The isolated or overlapping status of a given *self* DQ window depends on the category of the veto that *self* DQ would produce, as well as the category of the vetoes of *other* DQs whose windows overlap in time with the *self* window.

category 4 applies to *self* window, and category 5 applies to the *other* window which overlaps with *self*, then the *self* window is considered an *isolated* window. In CBC inspiral searches, a *self* window of category 2 would be considered *overlapping* if it overlapped in time with *other* window of category 3, since the category assigned to injections (`INJ_CAT`) of the search is category 3. However, a *self* window of category 2 would be considered *isolated* if it only overlapped in time with *other* windows of category 4 or 5.

Using the `OverlappingWindows` object, the main program then writes XML tables listing the isolated windows and overlapping windows. In the `MATLAB` scripts, the user was limited to examining only those DQ windows which overlapped in time with one other DQ window. `DQTunePipe` allows the user to set a maximum number of overlapping windows to examine, as the `OverlappingWindows` object keeps a record of all the *other* windows with which a single DQ window overlaps. The default behavior is to print to the XML table DQ windows with a maximum of three overlapping windows, thereby creating three XML isolatedTriggers_L1.xml

1: ifo	2: startTime	3: endTime	4: snrGreaterThan	5: MainWindowCategories	6: FileGeneratedAt
L1	968457615	968803215	60	[1; 2; 3; 4; 5]	Fri Nov 4 06:01:28 2011

ClusteredTriggers

1: Count	2: triggerTime	3: snr	4: mass	5: DQName	6: category	7: PaddedWindowStart	8: PaddedWindowEnd	9: Total (Unique) window periods
1	968580429	68.65725	99.99999	DCH-OMC_INPUT_ANGULAR_MED	4	968580428	968580432	1
2	968584081	87.64344	86.48267	DCH-OMC_INPUT_ANGULAR_MED	4	968584080	968584083	1
3	968619833	334.852	62.66738	DCH-CBC_HIGHMASS_SNR_GT_250	4	968619817	968619850	1
4	968629832	118.1772	100	DCH-ALL_SAFE_UPV	4	968629830	968629833	1
5	968633633	68.314	99.999994	DMT-INJECTION_BURST	2	968633584	968633744	1
6	968633637	81.66029	100	DMT-INJECTION_BURST	2	968633584	968633744	1
7	968633642	68.77889	65.04272	DMT-INJECTION_BURST	2	968633584	968633744	1
8	968649089	133.1454	85.230547	DCH-OMC_INPUT_ANGULAR_MED	4	968649089	968649091	1
9	968654549	121.7878	66.59704	DCH-INJECTION_INSPIRAL_BLIND	3	968654461	968654564	1
10	968704120	162.7563	40.96613	DCH-ALL_SAFE_UPV	4	968704118	968704121	1
11	968710572	178.2464	75.06149	DCH-OMC_INPUT_ANGULAR_MED	4	968710571	968710572	1
12	968723424	85.76736	69.99528	DCH-OMC_INPUT_ANGULAR_MED	4	968723423	968723425	1
13	968754163	80.54695	99.999996	DMT-INJECTION_BURST	2	968754121	968754272	1
14	968754168	104.0732	100	DMT-INJECTION_BURST	2	968754121	968754272	1
15	968754172	93.55256	74.43159	DMT-INJECTION_BURST	2	968754121	968754272	1
16	968763852	108.1881	88.542	DCH-OMC_INPUT_ANGULAR_MED	4	968763850	968763852	1

Figure 3.4: Example of Isolated Trigger Table - as displayed in browser with ligo lightweight stylesheet.

tables: an XML table listing DQ windows with two other overlapping windows, an XML table listing DQ windows with one other overlapping window, and an an XML table listing DQ windows with zero other overlapping windows, i.e. isolated DQ windows.

Next, this task identifies the isolated and overlapping triggers and writes them, again to an XML file. Only the triggers whose `snr` values are above a predetermined threshold, `SNR_ISOLATED_THRESH`, are evaluated. A trigger is considered *isolated* if it is occurs during a given DQ flag window that is also considered *isolated*. An *overlapping* trigger occurs during an *overlapping* DQ flag window, regardless of when the trigger occurs with respect to that window's duration. These triggers are plotted against the background of unclustered triggers, along with the boundaries of the DQ flag windows in which they occurred.

As with `FiguresOfMerit`, the XML tables produced by this task are designed to be viewed with `LIGO_LW_XSL`, which this task copies into the necessary subdirectories it creates. Configuration options allow the user to select whether to execute this task, as well

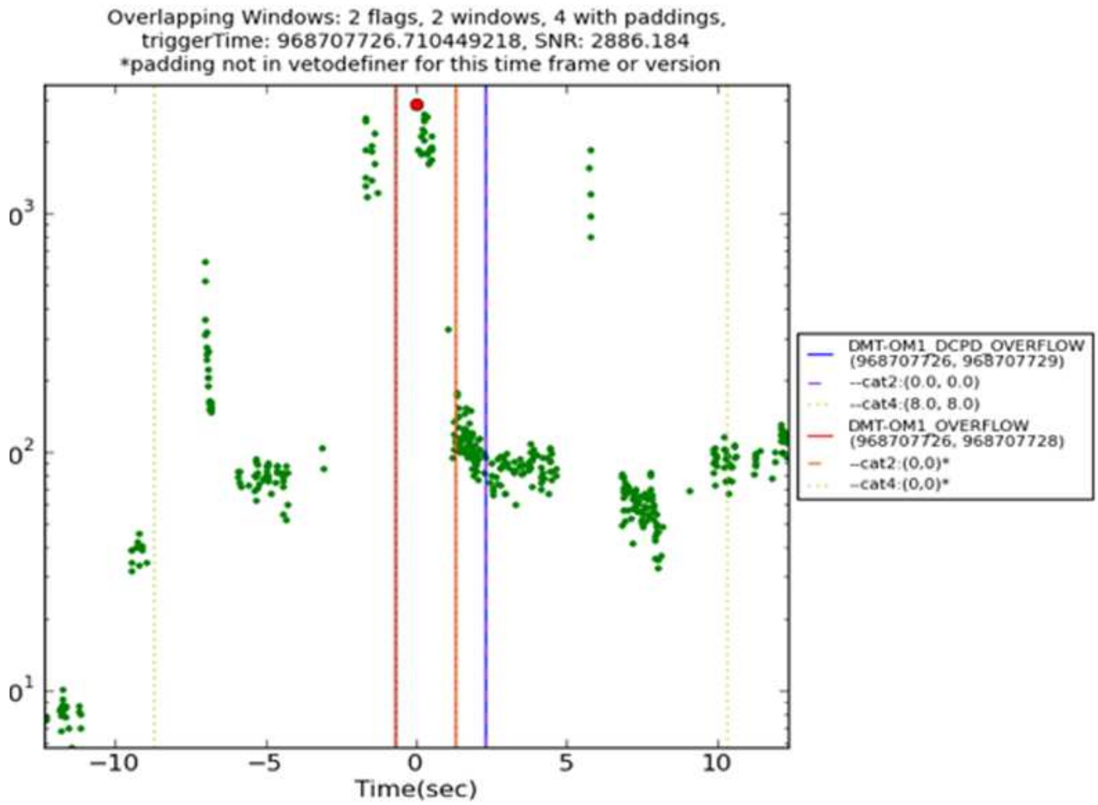


Figure 3.5: Example of Overlapping Trigger Plot - Overlapping trigger (larger dot, in red) plotted with background unclustered triggers and overlapping window boundaries.

as allow the user to choose only to write the isolated or overlapping window XML tables, or to create only the isolated or only the overlapping trigger plots. Since it is necessary to generate the `OverlappingWindows` object to create trigger plots, the command option to create either the overlapping (isolated) plots automatically activates the command to create the corresponding XML tables for the overlapping (isolated) windows and triggers. In contrast, choosing the option to create only the overlapping (isolated) XML tables will not create the corresponding plots.

3.5.3 Vetoed, Non-Vetoed, and Outlying Triggers

Another task of `DQTunePipe` is to determine at what category level triggers are vetoed. The `VetoedAndNonvetoedTriggers` object determines which triggers are vetoed at which category level, and creates a set of XML tables, listing the triggers vetoed and the triggers

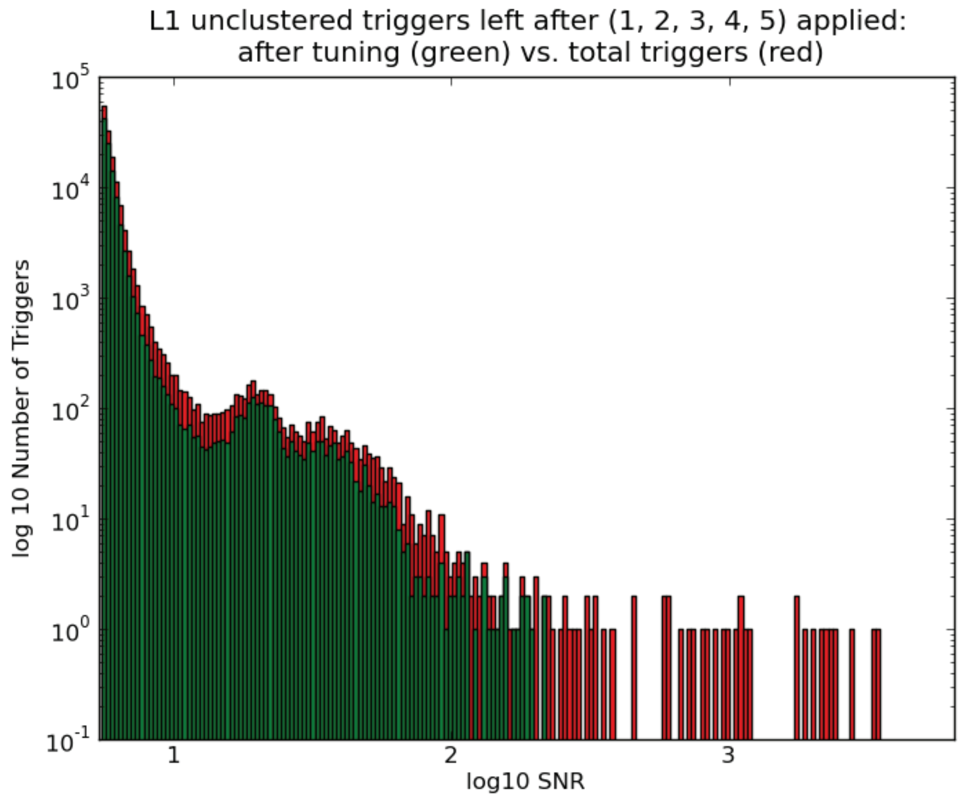


Figure 3.6: Example Histogram of vetoed and nonvetoed triggers

not vetoed, for each category level. In addition, `VetoedAndNonvetoedTriggers` will also use its `histogram` method to generate histograms identifying the vetoed and non-vetoed triggers at each category level (See Figure 3.6). `VetoedAndNonvetoedTriggers` task also uses the class `Outliers` to create a set of XML tables that lists the *outlying triggers*, i.e. non-vetoed triggers at each category level whose snr value is above the given `OUTLIER_THRESH` threshold, but occur *nearby* to `DQ` windows. A *nearby* `DQ` window is the `DQ` window whose boundary is both contained within the same analyzed segment window as the trigger, and is the nearest `DQ` window boundary to the trigger's time. The `Outliers` class also creates plots of the outlying triggers (See Figure 3.7).

Again the XML tables produced by this task are designed to be viewed with `LIGO_LW_XSL`, which `VetoedAndNonvetoedTriggers` copies into the necessary subdirectories it creates. Configuration options allow the user to choose to execute this task, and to what extent. If the user selects to create either the histograms or outlying trigger plots, the corresponding veto and non-vetoed trigger file will be created (See Figure 3.6); when the `Histogram` task is active, the table indicates where the associated histogram is located. For the outlying trigger option, the outlying trigger XML table will also be created.

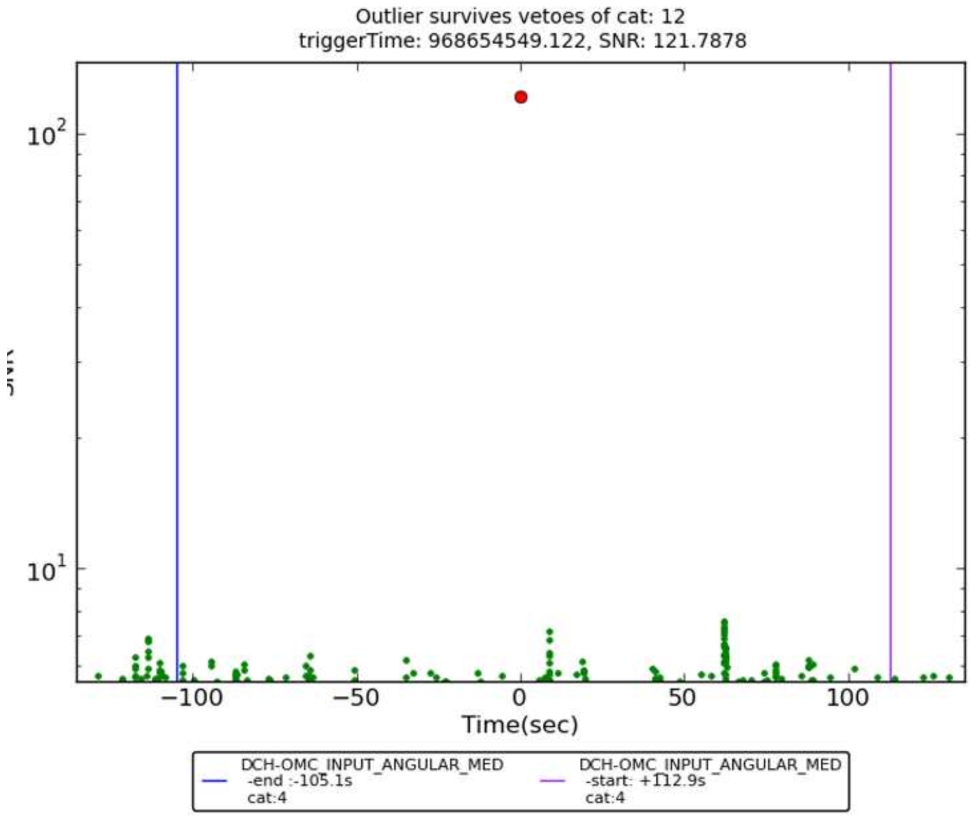


Figure 3.7: Example Outlier Plot - Outlying trigger (larger dot, in red) plotted against background unclustered triggers, with nearest DQ window boundary.

CHAPTER 4

USING THE DATA QUALITY TUNE PIPE

`DQTunePipe` is a flexible tool for DetChar analysis. This flexibility allows the user to run `DQTunePipe` in a number of different ways and configurations. Each of these varied executions of the program requires some degree of configuration by the user, although defaults are in place to reduce this configuration requirement to a minimum.

In its simplest format, the program may by invoked from the command line with only a single argument: the path to the configuration file.

```
> python DQTunePipe.py -f (configuration filename)
```

Figure 4.1

In addition to configuration parameters available to the user, DQTunePipe is also designed to be extensible for future development. By extending the existing DQTunePipe and taking advantage of the framework that it provides, the necessity of implementing common code structures such as property configuration, parsing data files, etc., is removed and a developer may instead focus on new functionality. Expected types of new functionality that a developer may add to the base program include new metrics and new tasks.

This chapter discusses the requirements for running the program, its various configuration properties, the default values of those properties, and how to set user-specified values for these properties via the command line and configuration file. The chapter will conclude with instructions on how to extend DQTunePipe by implementing a new metric and a new task.

4.1 Requirements

It is assumed that every user attempting to install and run this program will be operating inside of the LIGO Data Grid environment. As such, the user will have access to a version of Python $\geq$ 2.4 with accompanying NumPy and SciPy modules, and a current copy of LIGO tools (specifically `Glue`). When running, the program will expect to have available a number of input data files.

To install the program, the user may unpack the source, `DQTunePipe.tar.gz`, into a directory (for example `~/DQTunePipe/`). We will call this directory `$DQTunePipe_HOME`. Inside the directory the following Python source files should be present:

```
DQTunePipe.py                   outliers.py
config_rules.py                 OverlappingWindows.py
configuration.py                plotTriggers.py
createDataSet.py                properties.py
entities.py                     rawData.py
ExampleConfigurationFile.txt    summaryDataFiles.py
figuresOfMerit.py               utility_box.py
initialize.py                   vetoedAndNonVetoedTriggers.py
mod_ligolw.xsl                  writeDeadTime.py
manageData.py                   xmlColumnTag.py
metric.py                       XMLTableReader.py
```

In addition to the source files the necessary input data should be available to the program. `DQTunePipe` requires the input data be of the standard ***ligolw*** xml format, and includes:

1. an existing *vetodefiner* file (required when padding scenarios are to be investigated),

2. DQ xml files available via `ligolw_segment_query`, including the `DQ` flag that specifies the *science* data.

3. CBC first stage trigger xml files (either zipped or unzipped).

4.2 Configuration Option File

The user provides input parameters via the configuration file. The attributes of the `Properties` class identify these parameters. The configuration file is a simple text file with a list of the attributes of the `properties` object and the value for each attribute, separated by an equal sign. Figure (4.2) provides an example configuration file with the minimum required information that the user must provide. The example configuration file in Figure

```
INITIAL_START = 931035296
INITIAL_END = 935798487
OUTPUT_DIRECTORY = $HOME/OutputDir/
VETODEFINERFILE_FULLPATH = $HOME/vetodefiner.xml

#Either TRIGGER_SOURCE_DIRECTORIES or IHOPE_TRIGGER_SOURCE:
TRIGGER_SOURCE_DIRECTORIES=/archive/home/user/931035296-935798487/full_data/
IHOPE_TRIGGER_SOURCE = /archive/home/cbc/ihope_daily/

CLUSTERED_FORMAT =  -SIRE_FIRST_FULL
UNCLUSTERED_FORMAT =  -INSPIRAL_FIRST_FULL

#Name of DQ flags that denote science time: DQ_SCIENCE_H1, DQ_SCIENCE_H2,
#DQ_SCIENCE_L1, and DQ_SCIENCE_V1, unless only a single IFO is specified:
DQ_SCIENCE_H1 = -DMT_SCIENCE
DQ_SCIENCE_L1 = -DMT_SCIENCE
DQ_SCIENCE_H2 = -DMT_SCIENCE
DQ_SCIENCE_V1 = -ITF_SCIENCEMODE
```

Figure 4.2: Example of minimum configuration file - The attributes of the `Property` class, documented in Appendix B.1.3, and the values the user wishes to assign to each attribute are separated by an equal sign. A "#" denotes a commented line.

(4.2) illustrates how the configuration parameters, the attributes of the `Properties` class, are set by the user. The first three entries are `INITIAL_START` and `INITIAL_END`, the gps start and end times over which the DQ flags are evaluated, respectively, and `OUTPUT_DIRECTORY`, the output directory in which `DQTunePipe` writes the output.

`VETODEFINERFILE_FULLPATH` is the location of the vetodefiner file. If only unpadded data are to be evaluated, then `VETODEFINERFILE_FULLPATH` is not required. Otherwise, the user must supply the vetodefiner file's location. A local copy may be specified, as in the example, or a server location, such as `VETODEFINERFILE_FULLPATH = https://www.lsc-group.phys.uwm.edu/ligovirgo/cbc/public/segments/S6/vetofile.xml` can also be specified. `DQTunePipe` will then attempt to retrieve it via `wget`. In either case the entire full path location and name of the vetodefiner file must be provided.

`TRIGGER_SOURCE_DIRECTORIES` is the directory location of trigger xml files, zipped or unzipped. This parameter can also accept multiple directory names, comma-separated. Alternatively, the user can set the parameter for `IHOPE_TRIGGER_SOURCE`, which, if used, is expected to contain trigger xml files, zipped or unzipped, in sub-directories of form `IHOPE_TRIGGER_SOURCE/`*`yearMonth`*`/`*`yearMonthDay`*`/`. If values for both `TRIGGER_SOURCE_DIRECTORIES` and `IHOPE_TRIGGER_SOURCE` are provided, `DQTunePipe` will use `TRIGGER_SOURCE_DIRECTORIES` to locate triggers.

`CLUSTERED_FORMAT` and `UNCLUSTERED_FORMAT` identify the file name format for clustered and unclustered trigger files. `DQ_SCIENCE_H1`, `DQ_SCIENCE_H2`, `DQ_SCIENCE_L1`, and `DQ_SCIENCE_V1` identify the DQ flag naming scheme for *science* data for H1, H2, L1 and V1, respectively.

4.3 Additional Configuration Options

In addition to these requirements, a number of other properties may be specified by the user in the configuration file. These properties are listed in Figures 4.3a and 4.3b. If the value for the `properties` attribute is left blank, `DQTunePipe` will use default values. The default values of these configurable attributes, in conjunction with the values assigned by Figure 4.2, are shown in Figure 4.6.

```
#ADDITIONALLY AVAILABLE CONFIGURABLE PROPERTIES:
#List IFOs to use, accepts comma-separated list.
IFOS =

#List SNR thresholds to use, accepts comma-separated list.
THRESHOLDS =
#SNR thresholds to use for determining isolated triggers.
SNR_ISOLATED_THRESH =
#SNR threshold to use for determining outlying triggers.
OUTLIER_THRESH =

#Maximum number of overlapping DQ windows to consider.
MAX_OVERLAPS =
#Minimum number of overlapping DQ windows to consider. Advise: Must be > 0
MIN_OVERLAPS =

#Specify whether to calculate padded or unpadded Metrics (True or False):
PADDINGS =
#Specify whether to examine clustered or unclustered data (True or False):
IS_CLUSTERED =

#Do Tasks(True or False)- MAY BE PREFERABLE TO USE COMMAND LINE OPTIONS
DO_FOM_TABLE =
DO_DEAD_TIME_TABLE =
DO_ISOLATED =
DO_OVERLAPPING =
DO_ISOLATED_PLOTS =
DO_OVERLAPPING_PLOTS =
DO_VETOED_NONVETOED_TRIGGERS =
DO_OUTLIERS =
DO_HISTOGRAM =

#Print Values to be assigned to running environment and exit (True or False):
PRINT_VALUES =

#Specify gravitational-wave search (currently only supports "INSPIRAL")
# - for use in overlapping windows, applies category = 3 to injections
GW_PROGRAM =

#Specify specific DQ flags to be excluded, as regular expressions,
#in comma-separated list, for example: .*BCV\-.*,.*UPV\-.*
EXCLUDE_DQ_FLAGS =
```

Figure 4.3a: Example configuration file - *continued*: Since the running environment of `DQTunePipe` is configured by assigning values to the attributes of `properties`, the configuration file must be in the format illustrated by the examples in Figures 4.2, 4.3a, and 4.3b.

```
##ADDITIONALLY AVAILABLE CONFIGURABLE PROPERTIES (CONTINUED):
#Existing clustered and unclustered trigger summary files.
UNCLUST_TRIGGER_FILE_H1 =
UNCLUST_TRIGGER_FILE_H2 =
UNCLUST_TRIGGER_FILE_L1 =
UNCLUST_TRIGGER_FILE_V1 =
CLUST_TRIGGER_FILE_H1 =
CLUST_TRIGGER_FILE_H2 =
CLUST_TRIGGER_FILE_L1 =
CLUST_TRIGGER_FILE_V1 =
FULL_TRIGGER_SEGMENT_FILE_H1 =
FULL_TRIGGER_SEGMENT_FILE_H2 =
FULL_TRIGGER_SEGMENT_FILE_L1 =
FULL_TRIGGER_SEGMENT_FILE_V1 =

#Specify location to find DQ xml files (if not pulling from server)
DQ_XML_DIRECTORY =

#Specify ligo_lightweight style-sheet for viewing xmls (must be compatible)
LIGO_LW_XSL =
```

Figure 4.3b: Example configuration file - Additional configurable properties.

Properties that are most likely to be set differently between runs may also be set by a command line argument, see Figures 4.5a, 4.5b, and 4.5c. The user may assign all the configurable values in the configuration file, or choose some combination of command line options that include the required configuration file and other command-line options. For example, the user may wish to run each task individually using the same configuration file. The configuration file is always required, but all command line options except for the one which specifies the configuration file are optional. If the user specifies a property both in the configuration file and on the command line, the command line argument is used.

4.4 Command Line Configuration Options

Use of command line arguments allow the user to override parts of the configuration file. Not all configuration parameters are available via the command line, only those options which are likely to frequently vary. These include options that specify that `DQTunePipe` only executes selected tasks or evaluates only particular data types, and options that assist the user in operating `DQTunePipe`. All the command line options are described in the output of the help command, `-h` or `--help`, the output of which is seen in Figure (4.5a).

```
> python DQTunePipeControl -h
```

Figure 4.4

```
options:
  -h, --help            show this help message and exit
  REQUIRED:
    -f CONFIGFILE, --file=CONFIGFILE
            read configuration data from CONFIGFILE, must contain:
            INITIAL_START,
            INITIAL_END,
            OUTPUT_DIRECTORY,
            VETODEFINERFILE_FULLPATH (if padded data is to be evaluated),
            TRIGGER_SOURCE_DIRECTORIES or IHOPE_TRIGGER_SOURCE,
            CLUSTERED_FORMAT,
            UNCLUSTERED_FORMAT,
            DQ_SCIENCE_H1 (if H1 interferometer data is to be examined),
            DQ_SCIENCE_H2 (if H2 interferometer data is to be examined),
            DQ_SCIENCE_L1 (if L1 interferometer data is to be examined), and
            DQ_SCIENCE_V1 (if V1 interferometer data is to be examined).
```

Figure 4.5a: Output of help option - Available command line arguments, `-f` ***`configuration file`*** must be specified from command line. (*continues on next page*).

The user may specify that `DQTunePipe` uses data from a specific interferometer (`--ifo`), or specify SNR thresholds for figure of merit quantities (`--thresh`). The user may also specify subsets of data to be evaluated. `DQTunePipe` may calculate metric quantities on padded or unpadded DQ windows exclusively(`--pd` or `--unpd`). When `--unpd` is used, `DQTunePipe` does *not* retrieve the vetodefiner file, does not apply the padding and category information to the `dataSet` object, and does not execute tasks that require the padding information; only the `FiguresOfMerit` and `VetoedAndNonVetoedTriggers` tasks, for unpadded values, are executed. The user may also prefer to only evaluate clustered or unclustered trigger data (`--clustered-only` or `--unclustered-only`)

```
MORE CONFIGURATION OPTIONS:
  --ifo=IFO           indicate IFO, ex: H1 or H2 or L1 or V1.
  --thresh=THRESHOLDS, --threshold=THRESHOLDS
                      Use custom snr threshold values in determining
                      figure of merit tables.
  --unpd, --no-padding
                      when running pipeline, get figure of merit table
                      only for UNPADDED data quality windows.
                      DEFAULT: both padded and unpadded.
  --pd, --only-padding
                      when running pipeline, get figure of merit table
                      only for PADDED data quality windows.
                      DEFAULT: both padded and unpadded.

CLUSTERED DATA OPTIONS:
  Note: Use these options with the knowledge that increased data equals
  increased processing time.These options allow you choose to run
  pipeline on only clustered or unclustered data.
  --clustered-only    Uses clustered trigger data only, still retrieves
                      unclustered for later use
                      unless a start-Point option is also specified.
  --unclustered-only  Uses unclustered trigger data only, still retrieves
                      clustered for later use
                      unless a start-Point option is also specified.

HOUSE CLEANING OPTION:
  Caution: use this option at your own risk.
  Note: if data directories are already populated with data, that data
  will be used, hence the house cleaning option.
  --clean             Consider Yourself WARNED: This DELETES existing
                      outputdir and ALL contents contained within, which
                      may include outputs for ifos as well as all initial
                      data. Note: Configuration file must still be
                      specified, so output directory may be identified.

DEBUGGING OPTIONS:
  --debug             Print some debugging info.
  --printValues       Print values of configurable properties and exit
                      (forces DEBUG=False) Configuration file must still
                      be provided.
```

Figure 4.5b: Output of help option, continued - Available command line arguments, `--debug` and `--clean` cannot be specified by configuration file, must be specified via command line. (*continues on next page*).

```
TASK OPTIONS:
  Which tasks can you do? If no tasks are specified, all tasks
  (excluding the individual dead time table) are executed. These options
  allow you to run only a particular part of the pipeline.
  Note: if any of these options are selected, they overwrite (to do -
  True) any corresponding tasks assigned in configuration file.

  --fom                 Write figure of merit table and summary figure of
                        merit files only.
  --deadTime            Write deadTime table only.
  --isolated            Produce list of isolated windows and isolated
                        triggers.
  --overlapping         Produce list of overlapping windows
                        and overlapping triggers.
  --isolatedPlots       Create lists of isolated windows,
                        isolated triggers, and plots of those triggers.
  --overlappingPlots    Create lists of overlapping windows,
                        overlapping triggers, and plots of those triggers.
  --histograms          Create lists and histograms of vetoed and non-vetoed
                        triggers.
  --outliers            Create list and plots of outlying triggers.
```

Figure 4.5c: Output of help option - Available Command line options for executing *tasks*.

The command line options also include (`--clean`), which removes any existing data from previous executions of `DQTunePipe` in the output directory; a debugging option, (`--debug`), which writes a detailed log of `DQTunePipe` actions, as well as an option to print the current `properties` values (`--printValues`).

4.4.1 `--printValues`, `--debug`, and `--clean`

Not all attributes of the `Properties` class may be configured (since they are internal references), and attempts to configure these will be ignored. To help the user identify configurable properties, `--printValues` lists all configurable properties and the current values they would be set to given the supplied configuration file (See Figure 4.6). All attributes, configurable or otherwise, along with other debugging information, are written in `dqtunepipe.log` in the user's operating directory if `--debug` is invoked. If the `--debug` is not used, the `dqtunepipe.log` is still written but omits the full attribute listing; only

```
properties.CLUST_TRIGGER_FILE_H1 = $HOME/OutputDir/data/summary/clustered_time_snr_mass_H1.txt
properties.CLUST_TRIGGER_FILE_H2 = $HOME/OutputDir/data/summary/clustered_time_snr_mass_H2.txt
properties.CLUST_TRIGGER_FILE_L1 = $HOME/OutputDir/data/summary/clustered_time_snr_mass_L1.txt
properties.CLUST_TRIGGER_FILE_V1 = $HOME/OutputDir/data/summary/clustered_time_snr_mass_V1.txt
properties.CLUSTERED_FORMAT = -SIRE_FIRST_FULL
properties.CONFIGURATION_FILENAME = exampleConfigurationFile.txt
properties.DEBUG = False
properties.DO_DEAD_TIME_TABLE = False
properties.DO_FOM_TABLE = True
properties.DO_HISTOGRAM = True
properties.DO_ISOLATED = True
properties.DO_ISOLATED_PLOTS = True
properties.DO_OUTLIERS = True
properties.DO_OVERLAPPING = True
properties.DO_OVERLAPPING_PLOTS = True
properties.DO_VETOED_NONVETOED_TRIGGERS = True
properties.DQ_SCIENCE_H1 = DMT_SCIENCE
properties.DQ_SCIENCE_H2 = nil
properties.DQ_SCIENCE_L1 = DMT_SCIENCE
properties.DQ_SCIENCE_V1 = nil
properties.DQ_SERVER_LOCATION = ldbd://segdb.ligo.caltech.edu
properties.DQ_XML_DIRECTORY = $HOME/OutputDir/data/ALL_DQS_931035296-935798487/
properties.FULL_TRIGGER_SEGMENT_FILE_H1 = $HOME/OutputDir/data/summary/H1-segments.txt
properties.FULL_TRIGGER_SEGMENT_FILE_H2 = $HOME/OutputDir/data/summary/H2-segments.txt
properties.FULL_TRIGGER_SEGMENT_FILE_L1 = $HOME/OutputDir/data/summary/L1-segments.txt
properties.FULL_TRIGGER_SEGMENT_FILE_V1 = $HOME/OutputDir/data/summary/V1-segments.txt
properties.EXCLUDE_DQ_FLAGS = []
properties.GW_PROGRAM = INSPIRAL
properties.IFOS = [‘H1’, ‘L1’]
properties.IHOPE_TRIGGER_SOURCE = nil
properties.INITIAL_END = 935798487
properties.INITIAL_START = 931035296
properties.INJ_CAT = 3
properties.IS_CLUSTERED = [True, False]
properties.LIGO_LW_XSL = /mnt/zfs2/rankins/DQTunePipe/mod_ligolw.xsl
properties.MAX_OVERLAPS = 2
properties.MIN_OVERLAPS = 1
properties.OUTLIER_THRESH = 50
properties.OUTPUT_DIRECTORY = $HOME/OutputDir/
properties.PADDINGS = [True, False]
properties.PRINT_VALUES = True
properties.SNR_ISOLATED_THRESH = 60
properties.THRESHOLDS = [6, 8, 12, 20, 50, 100, 200, 500, 1000]
properties.TRIGGER_SOURCE_DIRECTORIES= [‘/archive/home/user/931035296-935798487/full_data/’]
properties.UNCLUST_TRIGGER_FILE_H1 = $HOME/OutputDir/data/summary/unclustered_time_snr_mass_H1.txt
properties.UNCLUST_TRIGGER_FILE_H2 = $HOME/OutputDir/data/summary/unclustered_time_snr_mass_H2.txt
properties.UNCLUST_TRIGGER_FILE_L1 = $HOME/OutputDir/data/summary/unclustered_time_snr_mass_L1.txt
properties.UNCLUST_TRIGGER_FILE_V1 = $HOME/OutputDir/data/summary/unclustered_time_snr_mass_V1.txt
properties.UNCLUSTERED_FORMAT = -INSPIRAL_FIRST_FULL
properties.VETODEFINERFILE_FULLPATH = $HOME/vetodefiner.xml
Exiting:  Finished Displaying Configurable Properties
```

Figure 4.6: Example of output from `--printValues` - configurable properties attributes and their assigned values are printed to the console. (Shown: default values in conjunction with `exampleConfigurationFile.txt` as in Figure 4.2, but with only `IFOS` = H1, L1 assigned.

status information and errors are written.

The local storage directory for triggers is not a configurable property and will always be

`OUTPUT_DIRECTORY/data/triggers-`*(individual ifo)*. When `DQTunePipe` initially runs, if either this directory or the `DQ_XML_DIRECTORY` is already populated with data then `DQTunePipe` does not attempt to retrieve this raw data again, as described in Section 3.2. Likewise, if the intermediate data is already located in `DATA_SUMMARY_DIRECTORY` (another non-configurable attribute) then both the intermediate stage and the retrieve data stage (for triggers) is skipped. Hence, `--clean` exists to remove any and all data from the supplied `OUTPUT_DIRECTORY`, which includes raw input, intermediate, and output data.

4.4.2 *Task* options

Among the most likely to be used of these command line options are the ***Task*** options. The ***Task*** options allow the user to indicate which specific tasks are to be completed by `DQTunePipe`. If *any* task option is selected, *all other non user-specified tasks default to non-active*, with the exception of other tasks upon which the selected task option is dependent.

4.5 Adding New Functionality

One of the advantages to `DQTunePipe` is the ability to add new functionality by employing the methods available to the `dataSet` and its related classes to analyze data. When incorporating new operations, the developer should first consider whether a new functionality requires its own class or should be incorporated into an existing class. This decision should be based both on the extent of the required calculations and the dependency relationship of those calculations to other tasks. For example, most of the metric quantities described in Section 2.1.4 are interrelated and computed by the methods of the `Metric` class. The exception to this is deadtime, which due to its simplicity and independence from other metric quantities, is calculated by a method of the `DQ` class, `deadTimeCalc` (Appendix C.1.3).

4.5.1 Including a New Metric Quantity

If a developer wants to incorporate a *new metric* quantity to be calculated into `DQTunePipe`, the necessary steps will be to incorporate the new quantity into the `Metric` (or possibly `DQ`) class by creating a new method for that class and calling it upon instantiation of the `Metric` class. Then the developer needs to append the `FiguresOfMerit` class to write that new quantity into the output xml files, and if desired, to the DQ metric summary txt files. Specifically, in the `FiguresOfMerit` class, the new metric quantity needs to be identified in the `writeMetricTable` method, and the values for the new quantity need to be included in the `writeMetricData` method. (See the example in Figure 4.7). To write to the summary text, the method `writeSummaryFile` in the `FiguresOfMerit` class needs to be modified in a similar manner to include the new quantity.

4.5.2 Writing New Tasks

If the new functionality is extensive enough to be considered a new *task*, (instead of a new metric quantity to be calculated) then it will require a new class and must also be incorporated into the main control structure. The steps to do this are:

1. Create the new class for the task.

   (a) Use existing methods of class `dataSet` to access the input information. Do not access data from the raw source, there is no need. The `dataSet` class (as well as its associated classes) is located in `entities.py`, see Appendix C.1.1.

   (b) Separate the output information from the task's calculations; either by creating a separate class to handle the output, or by creating a separate method for this task's class. This is done for consistency with existing tasks; future developers may wish to modify the output format.

   (c) Use the `properties.`*`attribute`* to access the properties to be used by this task; make certain that both `properties` and `log` have been imported from the

In class `Metric(object)`*, the metric quantity is first calculated when the object is initialized:*

```
...
#Calculate efficiency:
   if triggersAboveThresh == 0.0:
      efficiency = "NaN"
      self.__efficiency = float(efficiency)
   else:
      efficiency = ((float(self.totalTriggersVetoed))/
                  (float(triggersAboveThresh)))*100.0
      self.__efficiency = efficiency
...
#Calculate newMetricQuantity:
... DO STUFF TO CALCULATE THE NEW METRIC QUANTITY ...
    self.__newMetricQuantity = float(valueOfNewMetic)
```

Then, a method exists to reference the quantity:

```
  def getEfficiency(self):
    return self.__efficiency
  efficiency = property(getEfficiency)

  def getNewMetricQuantity(self):
    return self.__newMetricQuantity
  newMetricQuantity = property(getNewMetricQuantity)
```

In class `FiguresOfMerit`*: In method* ***`writeMetricTable`****, the metric quantities are identified, in the order in which they are included in the xml table, by* ***`dataTupleList`****.*

```
def writeMetricTable(self, ifo, properties):
...
  dataTupleList.append(("efficiency", "float"))
  dataTupleList.append(("newMetricQuantity", "float"))
...
```

Then, in method ***`writeMetricData`****, the values for the metric quantities are identified, in the order in which they are included in the xml table, by:*

```
def writeMetricData(self, ifo, threshold, dq, properties, doc, streamTag):
...
  line = line +","+str(round(metric.efficiency,6))
  line = line +","+str(round(metric.newMetricQuantity,6))
...
```

Figure 4.7: Example of including new metric quantity - This example presents code snippets to show how the metric quantity ***`efficiency`*** is evaluated and written to xml tables, and how to include a new metric quantity ***`newMetricQuantity`***. The new metric quantity is calculated and included as part of the `Metric` object, and then each `Metric` object is written to output by `FiguresOfMerit`.

`initialize` module so they are available for the new task to use.

2. Incorporate the new task into the main control structure.

   (a) Create a new attribute, `DO_`*`TASK`*, to notify `DQTunePipe` when to execute this task:

      i. In `Properties` class add a new attribute, `DO_`*`NEWTASK`*, following the format of other `DO_`*`NEWTASK`*-type attributes, as in illustrated in Figure (4.8)

```
def _get_do_newtask(self):
    return self._do_newtask
def _set_do_newtask(self, do):
    self._do_newtask = parseBoolean(do)
def _get_default_do_newtask(self):
    return True
DO_NEWTASK = property(_get_do_newtask, _set_do_newtask)
```

Figure 4.8: Adding a `DO_`*`NEWTASK`* attribute to the `Properties` class - all `Properties` attributes have `_get_` methods, attributes that are configurable by the user have `_set_` methods, and attributes with defined default values have `_get_default_` methods.

      i. In `parse_options()` (in `configuration.py`), add a line similar to Figure 4.9 to the `OptionGroup` called `taskOptions` to make this new task an executable command line option.

```
taskOptions.add_option("--newtask", action="store_true", dest="DO_NEWTASK",
                        help="Execute this new task.")
```

Figure 4.9: Creating a --newTask command-line option

   (b) In main control structure:

      i. Include a call to `properties.DO_`*`NEWTASK`* in main control structure to indicate whether to execute this new task.

      ii. Include a call to instantiate the new task's class, followed by a call to the output class or method for this task as well.

iii. Use `log.info(`*`message`*`)` (or *`log.debug(message)`*) to include status (or debugging) information in `dqtunepipe.log`.

Figure 4.10 illustrates how a ***NewTask*** *might* be incorporated into `DQTunePipe`.

```
if properties.DO_NEWTASK:
   task_string = ifo +": NEWTASK: "
   log.info(task_string + "BEGIN TASK")

   for isClustered in properties.IS_CLUSTERED:
      if isClustered:
         cluster_string = "clustered"
      else:
         cluster_string = "unclustered"

      for padding in properties.PADDINGS:
         if padding:
            padding_string = "padded"
         else:
            padding_string = "unpadded"
         log.info(task_string + "calculating for "+cluster_string
                  +" triggers and "+padding_string+" windows")

         #Create the NEWTASK object - initialize and calculates NEWTASK:
         NEWTASK_OBJECT = NEWTASK(dataSet, padding, isClustered)
         log.info(task_string + "calculation finished for "+cluster_string
                  +" triggers and "+padding_string+" windows")

         #Call NEWTASK's writeOutput method to generate output:
         log.info(task_string + "writing for "+cluster_string
                  +" triggers and "+padding_string+" windows")
         NEWTASK_OBJECT.writeOutput(ifo)
   log.info(task_string + "END TASK")
```

Figure 4.10: Example of control structure calling a *NewTask*, with logging. The control structure will only execute task if the value of `properties.DO_`*`NEWTASK`* is `True`. This example assumes that *`NewTask`* would be executed for both padded and unpadded values, on both clustered and unclustered data. To execute on only padded data, replace "`for padding in properties.PADDINGS:`" with "`if filter(None, properties.PADDINGS):`" or, for only unpadded data, with "`if not filter(None, properties.PADDINGS):`" (Similar modifications can be made with `properties.IS_CLUSTERED` to execute on only clustered or unclustered data).

4.5.3 Extending `DQTunePipe` to non-inspiral searches

In addition to adding new tasks, `DQTunePipe` can be extended to evaluate DQ flags applicable to other gravitational wave searches, provided those searches rely on similar trigger data to identify potential gravitational waves. Specifically, the trigger data must consist of triggers identifiable at instants of time with corresponding SNR values to evaluate the relationship between the triggers and DQ flags. For the purposes of extending `DQTunePipe` to other searches, a configurable property attribute `GW_PROGRAM` has been included, though at present it is not used as the `DQTunePipe` currently only supports inspiral gravitational wave searches. While it is beyond the scope of this thesis to cover in depth the requirements of extending `DQTunePipe` to cover non-inspiral searches, all such extension will likely require the modification of a few modules.

In `properties.py`, the new program name must be added to the collection of `supported_gw_programs`. To ensure the correct trigger data is evaluated, `summaryDataFiles.py` must be extended to appropriately support both the existing `GW_PROGRAM == "INSPIRAL"` and the new program value. Both the `make_intermediate_trigger_data` and `time_and_snr` functions look for specific tag names to identify the information in the XML files, which may not be applicable to non-inspiral searches. If new functions are added to generate intermediate trigger data, then in `DQTunePipe.py` the correct `make_intermediate_trigger_data` function must be called for the correspondingly supported `GW_PROGRAM`. It may also be necessary to limit which tasks are applicable to select programs, this may be done in `configuration.py`, see the section of the code identified as *verify tasks*.

4.6 Maintaining `DQTunePipe`

A new task can be added by creating a new object class and instantiating the object from `DQTunePipe`'s main control structure. It is uncomplicated to incorporate a new task, as well as modify an existing task, because the abstraction layer of the `DataSet` class buffers the task execution from the raw input data. A configurable attribute, `GW_PROGRAM`, is also already in place to aid the developer in extending `DQTunePipe` to other searches.

`DQTunePipe` is also highly configurable, due to the utilization of `Properties` attributes. The attributes of the `Properties` class with `_set_` methods afford the user the luxury of customizing `DQTunePipe`'s environment through the use of the configuration file and command-line arguments. Yet attributes with `_get_default_` methods keep the user's actual input requirements to a minimum. Likewise, attributes with `_get_` methods, along with the abstraction layer of the `DataSet`, make adding new functionality to `DQTunePipe` a straight-forward endeavor.

CHAPTER 5

EXAMPLE ANALYSIS

LIGO scientists may use the output of `DQTunePipe` to tune DQ flags in order to improve the effectiveness of vetoes produced by those flags. Tuning may take the form of adjusting the padding applied to windows of a DQ flag.

In this chapter, we will present a brief analysis of the results of applying `DQTunePipe` to data collected over the four day time span, September 14-18, 2010, 00:00:00 UTC, applied to highmass inspiral triggers, configured as indicated by Figure 5.1. In particular we will look at a few flags in detail: for the H1 interferometer, we will examine H1:DMT-OM1_DCPD_OVERFLOW, H1:DMT-OM1_OVERFLOW, and H1:DMT-SEVERE_LSC_OVERFLOW. These three are all overflow flags, as described in Section 2.1.1. These flags were eventually chosen for use as category 2 and category 4 vetoes. For the L1 interferometer, we will briefly examine L1:APC-L0_PEM_BSC4_MIC, L1:APC-L0_PEM_BSC5_MIC, L1:APC-L0_PEM_HAM6_MIC, L1:APC-L0_PEM_ISCT1_MIC, and L1:APC-L0_PEM_LVEA_MIC, and discuss how the their figure of merit values lead us to determine that they should not be considered as vetoes. For the purposes of this analysis, we assume we do not know the category assignments.

```
INITIAL_START = 968457615
INITIAL_END = 968803215
OUTPUT_DIRECTORY = /home/rankins/WWW/BIGDOG_20100914-20100918

VETODEFINERFILE_FULLPATH =
           https://www.lsc-group.phys.uwm.edu/ligovirgo/cbc/public/
           segments/S6/H1L1V1-S6_CBC_HIGHMASS_D_OFFLINE-961545543-0.xml

TRIGGER_SOURCE_DIRECTORIES =
           /home/tdent/S6/S6d/highmass/chunk3_20110310/
           968543943-971622087/full_data

UNCLUSTERED_FORMAT = -INSPIRAL_FIRST_FULL
CLUSTERED_FORMAT = -SIRE_FIRST_FULL

DQ_SCIENCE_H1 = -DMT-SCIENCE
DQ_SCIENCE_L1 = -DMT-SCIENCE

MAX_OVERLAPS = 5
```

Figure 5.1: Configuration file used in example analysis - From September 14, 2010 00:00:00 UTC (gps time 968457615) to September 18, 2010 00:00:00 UTC (gps time 968803215).

5.1 Figures of Merit

As described in Chapter 3, the output of the `FigureOfMerit` task are the metric quantities of the flags defined in the time interval from INITIAL_START to INITIAL_END. These metric quantities will be used to verify the category assignments of the DQ flags that we are considering. The figure of merit of the three overflow flags are shown in Figure 5.2. The deadtime for these flags is smaller than the average deadtime of all other category 2 DQ flags over this period (See Figure 5.3). The efficiency per deadtime and the χ^2 value at SNR $= 6$ are comparable to the corresponding average quantities of Figure 5.3. Moreover, from Figure 5.4, we see that the maximum values of the χ^2 occur at high SNR thresholds. This is consistent with the results in Figure 5.3, where it seen that χ^2 values peak at high SNR. All these observations suggest that these flags should be assigned as category 2.

A similar analysis can be performed for the L1 microphone flags:

identifierTable

1: ifo	2: startTime	3: endTime	4: snr	5: triggers-Clustered	6: FileGeneratedAt
H1	968457615	968803215	6	13675	Wed Nov 23 00:28:32 2011

DQDataTable-UNPD

1: ifo	2: DQname	3: #of windows	4: SNR	5: deadTime (percentage)	6: efficiency (percentage)	7: usePercent	8: efficiency/ deadTime	9: chiSquared Value
H1	DMT-OM1_DCPD_OVERFLOW	13	6	0.0092	0.041	15.385	4.427	24.493
H1	DMT-OM1_OVERFLOW	61	6	0.062	0.408	31.148	6.572	369.335
H1	DMT-SEVERE_OM1_OVERFLOW	15	6	0.01	0.041	13.333	4.006	24.54

Figure 5.2: Figure of merit for H1:DMT-OM1_DCPD_OVERFLOW, H1:DMT-OM1_OVERFLOW, and H1:DMT-SEVERE_LSC_OVERFLOW - The values in the table are calculated using clustered triggers and unpadded DQ windows.

identifierTable

1: ifo	2: startTime	3: endTime	4: triggers-clustered	5: category	6: number of DQ Windows in category	7: FileGeneratedAt
H1	968457615	968803215	13675	2	333	Wed Nov 23 00:06:19 2011

DQDataTable-PD: excludes from calculation: values that are 0 nor nan

1: threshold	2: mean deadTime (percentage)	3: mean efficiency (percentage)	4: mean usePercentage	5: mean efficiency/deadTime	7: mean chiSquared
6	0.839	0.296	85.714	7.215	48.714
8	0.839	4.078	57.143	1.463	66.434
12	0.839	11.215	42.857	4.024	191.807
20	0.839	15.789	42.857	5.665	204.041
50	0.839	25	42.857	8.97	271.38
100	0.839	9.091	28.571	3.262	19.413
200	0.839	nan	nan	0	0.117
500	0.839	nan	nan	0	0.117
1000	0.839	nan	nan	0	0.101

Figure 5.3: Average metrics for all category 2 DQ flags, excluding the analyzed overflow flags.

L1:APC-L0_PEM_BSC4_MIC, L1:APC-L0_PEM_BSC5_MIC, L1:APC-L0_PEM_HAM6_MIC, L1:APC-L0_PEM_ISCT1_MIC, and L1:APC-L0_PEM_LVEA_MIC. Figure 5.5 shows the figure of merit summary detail for each of these flags. The maximum χ^2 values and the efficiency per deadtime values are comparable with the corresponding average values for category 5. The use percentage decreases with increasing SNR, which implies that high SNR glitches are not likely to be associated with these DQ flags. These results suggest that these flags should be used at best as category 5 vetoes. In S6, the DetChar group did not use these DQ flags as vetoes.

Output for IFO: H1 and DQ Flag: DMT-OM1_DCPD_OVERFLOW
Time analyzed: 206016.0sec
Number of inspiral clusters: 13675
Mean time between clusters: 15.065
Median snr of inspiral clusters: 5.845
Maximum snr of inspiral clusters: 4976.165
Veto window buffer: No Padding Applied

Number of veto windows: 13
Median length of veto windows: 1.0
Deadtime: 0.009%
Max chiSquared Value (SNR, chiSquared): at SNR > (1000, 11441.334)

SNR >	vetoed/total	Efficiency	Eff/deadTime	Used Windows/Total	UsedPercent
6	2/4898	0.04 %	4.43	2/13	15.38 %
8	2/613	0.33 %	35.38	2/13	15.38 %
12	2/214	0.93 %	101.34	2/13	15.38 %
20	2/114	1.75 %	190.23	2/13	15.38 %
50	2/60	3.33 %	361.43	2/13	15.38 %
100	2/22	9.09 %	985.72	2/13	15.38 %
200	2/14	14.29 %	1548.99	2/13	15.38 %
500	2/14	14.29 %	1548.99	2/13	15.38 %
1000	2/12	16.67 %	1807.16	2/13	15.38 %

Output for IFO: H1 and DQ Flag: DMT-OM1_OVERFLOW
Time analyzed: 206016.0sec
Number of inspiral clusters: 13675
Mean time between clusters: 15.065
Median snr of inspiral clusters: 5.845
Maximum snr of inspiral clusters: 4976.165
Veto window buffer: No Padding Applied

Number of veto windows: 61
Median length of veto windows: 2.0
Deadtime: 0.062%
Max chiSquared Value (SNR, chiSquared): at SNR > (1000, 17164.007)

SNR >	vetoed/total	Efficiency	Eff/deadTime	Used Windows/Total	UsedPercent
6	20/4898	0.41 %	6.57	19/61	31.15 %
8	12/613	1.96 %	31.51	11/61	18.03 %
12	6/214	2.80 %	45.13	6/61	9.84 %
20	2/114	1.75 %	28.24	2/61	3.28 %
50	2/60	3.33 %	53.65	2/61	3.28 %
100	2/22	9.09 %	146.32	2/61	3.28 %
200	2/14	14.29 %	229.93	2/61	3.28 %
500	2/14	14.29 %	229.93	2/61	3.28 %
1000	2/12	16.67 %	268.25	2/61	3.28 %

Output for IFO: H1 and DQ Flag: DMT-SEVERE_LSC_OVERFLOW
Time analyzed: 206016.0sec
Number of inspiral clusters: 13675
Mean time between clusters: 15.065
Median snr of inspiral clusters: 5.845
Maximum snr of inspiral clusters: 4976.165
Veto window buffer: No Padding Applied

Number of veto windows: 14
Median length of veto windows: 1.0
Deadtime: 0.013%
Max chiSquared Value (SNR, chiSquared): at SNR > (200, 9806.287)

SNR >	vetoed/total	Efficiency	Eff/deadTime	Used Windows/Total	UsedPercent
6	2/4898	0.04 %	3.24	2/14	14.29 %
8	2/613	0.33 %	25.85	2/14	14.29 %
12	2/214	0.93 %	74.05	2/14	14.29 %
20	2/114	1.75 %	139.01	2/14	14.29 %
50	2/60	3.33 %	264.12	2/14	14.29 %
100	2/22	9.09 %	720.34	2/14	14.29 %
200	2/14	14.29 %	1131.96	2/14	14.29 %
500	2/14	14.29 %	1131.96	2/14	14.29 %
1000	1/12	8.33 %	660.31	1/14	7.14 %

Figure 5.4: Detailed summary of figures of merit for H1:DMT-OM1_DCPD_OVERFLOW, H1:DMT-OM1_OVERFLOW, and H1:DMT-SEVERE_LSC_OVERFLOW.

5.2 Window Padding

It is of interest to examine the overflow flags with categories and padding applied. Because these flags are all overflows they were categorized and padded identically. The vetodefiner shows that these flags were also classified as category 4, with a padding of 8 seconds added to the beginning and end of each window. This dual categorization is typical of overflow flags because of the presence of additional noise preceding and following the flagged time.

Since no additional padding is applied at category 2, the metric quantities of the category 2 flags are identical to the metric quantities previously discussed. The metrics of the resulting category 4 veto are show in Figure 5.6.

There are two points of particular interest in Figure 5.6. First the number of windows for H1:DMT-OM1_DCPD_OVERFLOW and H1:DMT-SEVERE_LSC_OVERFLOW for category 2 is different from the number of windows of category 4. This can be explained as follows: The metrics are computed over science time, i.e. the DQ flag windows are intersected with H1:DMT-SCIENCE. When extra padding is considered, a window may overlap with an additional science segment, thus creating an extra DQ window in science time.

For instance, H1:DMT-OM1_DCPD_OVERFLOW has a window from gps time 968459019 to 968459022. This time interval is not in science time. However, when category 4 padding is applied, the padded window is defined from gps time 968459011 to 968459030 and now intersects the science time segment defined from gps time 968457615 to

```
Output for IFO: L1 and DQ Flag: APC-L0_PEM_BSC4_MIC
Time analyzed: 254148.0sec
Number of inspiral clusters: 18395
Mean time between clusters: 13.816
Median snr of inspiral clusters: 5.938
Maximum snr of inspiral clusters: 6574.139
Veto window buffer: No Padding Applied

Number of veto windows: 5
Median length of veto windows: 72.0
Deadtime: 0.16%
Max chiSquared Value (SNR, chiSquared):  at SNR > (8, 34.244)
```

SNR >	vetoed/total	Efficiency	Eff/deadTime	Used Windows/Total	UsedPercent
6	19/8136	0.23 %	1.46	5/5	100.00 %
8	2/544	0.37 %	2.30	1/5	20.00 %
12	1/286	0.35 %	2.19	1/5	20.00 %
20	0/187	0.00 %	0.00	0/5	0.00 %
50	0/85	0.00 %	0.00	0/5	0.00 %
100	0/51	0.00 %	0.00	0/5	0.00 %
200	0/24	0.00 %	0.00	0/5	0.00 %
500	0/15	0.00 %	0.00	0/5	0.00 %
1000	0/11	0.00 %	0.00	0/5	0.00 %

```
Output for IFO: L1 and DQ Flag: APC-L0_PEM_BSC5_MIC
Time analyzed: 254148.0sec
Number of inspiral clusters: 18395
Mean time between clusters: 13.816
Median snr of inspiral clusters: 5.938
Maximum snr of inspiral clusters: 6574.139
Veto window buffer: No Padding Applied

Number of veto windows: 6
Median length of veto windows: 106.0
Deadtime: 0.238%
Max chiSquared Value (SNR, chiSquared):  at SNR > (6, 84.038)
```

SNR >	vetoed/total	Efficiency	Eff/deadTime	Used Windows/Total	UsedPercent
6	54/8136	0.66 %	2.79	6/6	100.00 %
8	0/544	0.00 %	0.00	0/6	0.00 %
12	0/286	0.00 %	0.00	0/6	0.00 %
20	0/187	0.00 %	0.00	0/6	0.00 %
50	0/85	0.00 %	0.00	0/6	0.00 %
100	0/51	0.00 %	0.00	0/6	0.00 %
200	0/24	0.00 %	0.00	0/6	0.00 %
500	0/15	0.00 %	0.00	0/6	0.00 %
1000	0/11	0.00 %	0.00	0/6	0.00 %

```
Output for IFO: L1 and DQ Flag: APC-L0_PEM_LVEA_MIC
Time analyzed: 254148.0sec
Number of inspiral clusters: 18395
Mean time between clusters: 13.816
Median snr of inspiral clusters: 5.938
Maximum snr of inspiral clusters: 6574.139
Veto window buffer: No Padding Applied

Number of veto windows: 1
Median length of veto windows: 130.0
Deadtime: 0.051%
Max chiSquared Value (SNR, chiSquared):  at SNR > (6, 0.812)
```

SNR >	vetoed/total	Efficiency	Eff/deadTime	Used Windows/Total	UsedPercent
6	6/8136	0.07 %	1.44	1/1	100.00 %
8	0/544	0.00 %	0.00	0/1	0.00 %
12	0/286	0.00 %	0.00	0/1	0.00 %
20	0/187	0.00 %	0.00	0/1	0.00 %
50	0/85	0.00 %	0.00	0/1	0.00 %
100	0/51	0.00 %	0.00	0/1	0.00 %
200	0/24	0.00 %	0.00	0/1	0.00 %
500	0/15	0.00 %	0.00	0/1	0.00 %
1000	0/11	0.00 %	0.00	0/1	0.00 %

```
Output for IFO: L1 and DQ Flag: APC-L0_PEM_ISCT1_MIC
Time analyzed: 254148.0sec
Number of inspiral clusters: 18395
Mean time between clusters: 13.816
Median snr of inspiral clusters: 5.938
Maximum snr of inspiral clusters: 6574.139
Veto window buffer: No Padding Applied

Number of veto windows: 6
Median length of veto windows: 76.0
Deadtime: 0.213%
Max chiSquared Value (SNR, chiSquared):  at SNR > (6, 44.01)
```

SNR >	vetoed/total	Efficiency	Eff/deadTime	Used Windows/Total	UsedPercent
6	37/8136	0.45 %	2.13	6/6	100.00 %
8	2/544	0.37 %	1.72	2/6	33.33 %
12	2/286	0.70 %	3.28	2/6	33.33 %
20	2/187	1.07 %	5.02	2/6	33.33 %
50	0/85	0.00 %	0.00	0/6	0.00 %
100	0/51	0.00 %	0.00	0/6	0.00 %
200	0/24	0.00 %	0.00	0/6	0.00 %
500	0/15	0.00 %	0.00	0/6	0.00 %
1000	0/11	0.00 %	0.00	0/6	0.00 %

```
Output for IFO: L1 and DQ Flag: APC-L0_PEM_HAM6_MIC
Time analyzed: 254148.0sec
Number of inspiral clusters: 18395
Mean time between clusters: 13.816
Median snr of inspiral clusters: 5.938
Maximum snr of inspiral clusters: 6574.139
Veto window buffer: No Padding Applied

Number of veto windows: 3
Median length of veto windows: 132.0
Deadtime: 0.215%
Max chiSquared Value (SNR, chiSquared):  at SNR > (6, 18.357)
```

SNR >	vetoed/total	Efficiency	Eff/deadTime	Used Windows/Total	UsedPercent
6	29/8136	0.36 %	1.66	2/3	66.67 %
8	0/544	0.00 %	0.00	0/3	0.00 %
12	0/286	0.00 %	0.00	0/3	0.00 %
20	0/187	0.00 %	0.00	0/3	0.00 %
50	0/85	0.00 %	0.00	0/3	0.00 %
100	0/51	0.00 %	0.00	0/3	0.00 %
200	0/24	0.00 %	0.00	0/3	0.00 %
500	0/15	0.00 %	0.00	0/3	0.00 %
1000	0/11	0.00 %	0.00	0/3	0.00 %

Figure 5.5: Detailed summary of figures of merit for L1:APC-L0_PEM_BSC4_MIC, L1:APC-L0_PEM_BSC5_MIC, L1:APC-L0_PEM_HAM6_MIC, L1:APC-L0_PEM_ISCT1_MIC, and L1:APC-L0_PEM_LVEA_MIC

968459018. The interval from gps time 968459011 to 968459018 is now included in the evaluation of the figures of merit.

The second point of interest in the padded figure of merit table is that the metric calculations for H1:DMT-OM1_OVERFLOW for both categories 2 and 4 are identical. This is because the vetodefiner file only defines the categories and padding for this flag for a time range that ends before `INITIAL_START`. The category column in Figure 5.6 shows the relevant time frame over which the category and padding is applicable. If multiple paddings are defined for a single category, or the applicable time range is outside the user-defined `INITIAL_START` or `INITIAL_END` time, the vetodefiner file is linked in the table for reference.

identifierTable

1: ifo	2: startTime	3: endTime	4: snr	5: triggers-Clustered	6: FileGeneratedAt
H1	968457615	968803215	6	13675	Fri Nov 18 04:46:08 2011

DQDataTable-PD

1: ifo	2: DQname	3: category	4: #of windows	5: SNR	6: deadTime (percentage)	7: efficiency (percentage)	8: usePercent	9: efficiency/ deadTime	10: chiSquared Value
H1	DMT-OM1_DCPD_OVERFLOW	category=2; Lpad=0.0; Rpad=0.0; applicable=(Window[954000015.0; 968803215.0; 1])	13	6	0.0092	0.041	15.385	4.427	24.493
H1	DMT-OM1_DCPD_OVERFLOW	category=4; Lpad=8.0; Rpad=8.0; applicable=(Window[954000015.0; 968803215.0; 1])	27	6	0.13	0.204	14.815	1.593	47.22
H1	DMT-OM1_OVERFLOW	category=2 WARNING: padding not defined in vetodefiner for this time frame.	61	6	0.062	0.408	31.148	6.572	369.335
H1	DMT-OM1_OVERFLOW	category=4 WARNING: padding not defined in vetodefiner for this time frame.	61	6	0.062	0.408	31.148	6.572	369.335
H1	DMT-SEVERE_LSC_OVERFLOW	category=2; Lpad=0.0; Rpad=0.0; applicable=(Window[937473702.0; 968803215.0; 1])	14	6	0.013	0.041	14.286	3.235	24.659
H1	DMT-SEVERE_LSC_OVERFLOW	category=4; Lpad=8.0; Rpad=8.0; applicable=(Window[937473702.0; 968803215.0; 1])	24	6	0.11	0.102	8.333	0.91	22.046

Figure 5.6: Figure of merit for H1:DMT-OM1_DCPD_OVERFLOW, H1:DMT-OM1_OVERFLOW, and H1:DMT-SEVERE_LSC_OVERFLOW - The values in the table are calculated using clustered triggers and padded DQ windows.

5.3 Isolated and Overlapping Triggers

The analysis shows that there is an isolated trigger with SNR greater than 60. This trigger, at gps time 968744747.83593750 is flagged by a H1:DMT-SEVERE_OVERFLOW and is shown in Figure 5.8.

There are 12 additional isolated triggers identified over the evaluation period, and most of those triggers are contained within injection flags, seen in Figure 5.7. The figure of merit summaries, in Figure 5.4, for the three overflow flags we are evaluating, shows that each flag vetoes at least 2 triggers with SNR greater than 60. Since only one of these triggers is included in the isolated trigger table (for H1:DMT-SEVERE_LSC_OVERFLOW), there must be at least two triggers with SNR greater than 60 that are flagged by one of the three overflows and another category 2 DQ flag.

Indeed, the `DQTunePipe` analysis shows that there are two overlapping triggers for H1:DMT-OM1_DCPD_OVERFLOW and H1:DMT-OM1_OVERFLOW, and a third overlapping trigger, at 968778029.834 for all three overflow DQ flags. These are shown in Figures 5.9 and 5.10, respectively.

isolatedTriggers_H1.xml

1: ifo	2: startTime	3: endTime	4: snrGreaterThan	5: MainWindowCategories	6: FileGeneratedAt
H1	968457615	968803215	60	[1; 2; 3; 4; 5]	Thu Nov 3 03:13:57

ClusteredTriggers

1: Count	2: triggerTime	3: snr	4: mass	5: DQName	6: category	7: PaddedWindowStart	8: PaddedWindowEnd	9: Total (Unique) window periods
1	968587755	87.04017	99.999994	DMT-INJECTION_BURST	2	968587709	968587862	1
2	968587760	101.1961	86.08001	DMT-INJECTION_BURST	2	968587709	968587862	1
3	968587764	88.33353	65.27734	DMT-INJECTION_BURST	2	968587709	968587862	1
4	968633633	84.08903	99.99999	DMT-INJECTION_BURST	2	968633589	968633742	1
5	968633637	99.37577	99.99999	DMT-INJECTION_BURST	2	968633589	968633742	1
6	968633642	88.96572	59.40501	DMT-INJECTION_BURST	2	968633589	968633742	1
7	968744748	969.7728	99.99999	DMT-SEVERE_LSC_OVERFLOW	2	968744747	968744749	1
8	968754163	86.65681	99.999989	DMT-INJECTION_BURST	2	968754116	968754277	1
9	968754168	106.5191	99.99999	DMT-INJECTION_BURST	2	968754116	968754277	1
10	968754172	88.20905	59.68602	DMT-INJECTION_BURST	2	968754116	968754277	1
11	968774707	67.59113	66.060367	DMT-INJECTION_INSPIRAL	3	968774652	968774781	1
12	968774723	78.98104	99.999989	DMT-INJECTION_INSPIRAL	3	968774652	968774781	1
13	968797329	69.09262	31.88304	DCH-ALL_SAFE_UPV	4	968797326	968797334	1

Figure 5.7: List of isolated triggers with SNR greater than 60.

Incidentally, `DQTunePipe` also identifies an additional overlapping trigger, at 968605409.935 for three category 4 flags: H1:DCH-ALL_SAFE_UPV, H1:DMT-BRMS_SEISMIC_LVEA_Z_3_10_HZ_THRESH_2E3, and H1:DCH-SEISVETO_CBC. Note that there are five individual windows belonging to only three DQ flags in the interval when the overlap occurs, and this is indicated in the last column in Figure 5.10 where the total number of unique windows, 5 for this trigger, is shown.

The plots of the overlapping and isolated triggers illustrate to the user the applied padding amounts and may be used to determine if the chosen padding is appropriate. For instance, the plots of the overflow flags show that category 2 assignment vetoes the triggers of high SNR, but does not veto all triggers associated with the glitch. The padding applied to the flags at category 4 catches these additional low SNR triggers.

The user may set the number of overlapping flags in the analysis. This is useful in examining the relationship between flags. For example, Figure 5.11 shows a plot which is obtained by setting the number of overlapping flags to at least 5. Often, there is a well-understood relationship between some of the flags. For example, the flags in our 2 and 3 overlap examples are overflow flags and are generally expected to be active at the same

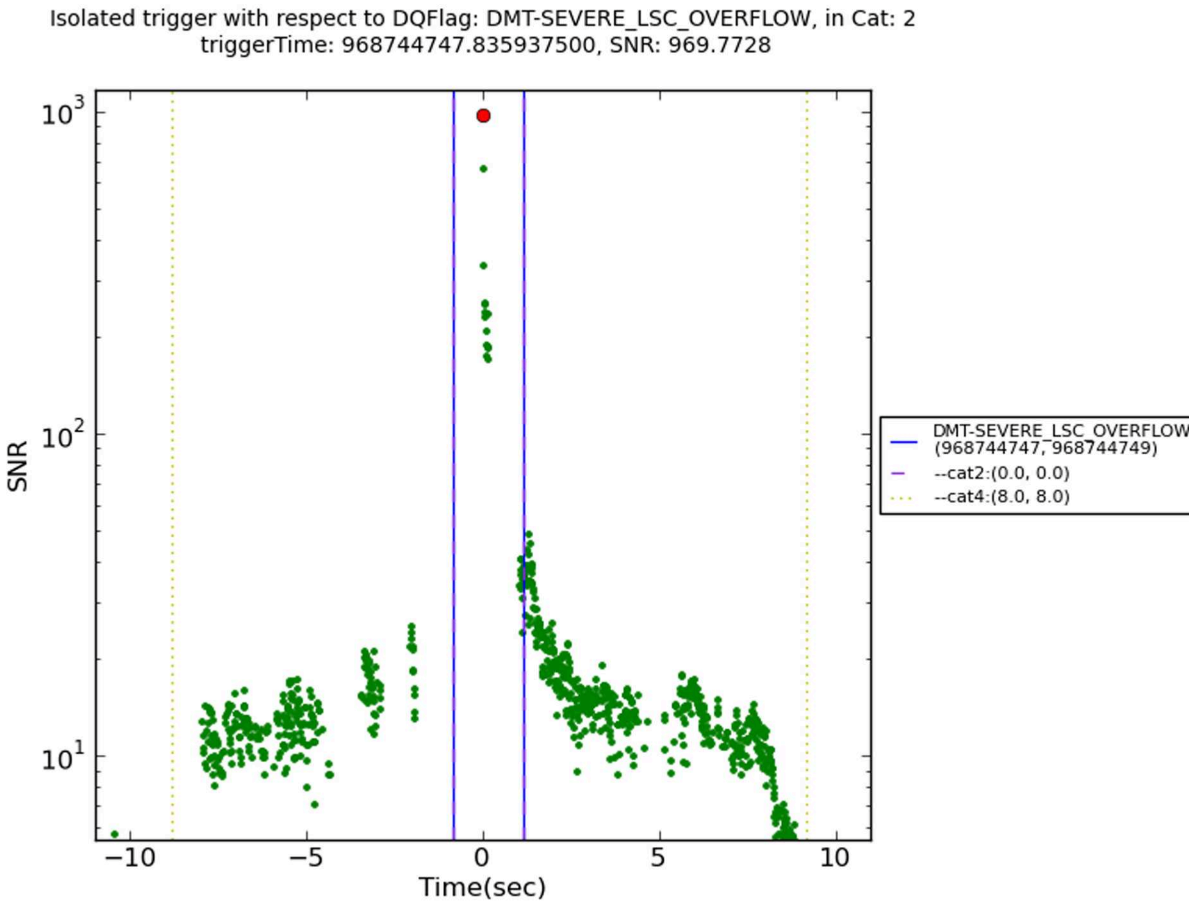


Figure 5.8: Plot of isolated trigger at gps time 968744747.83593750 - This trigger is contained within a window of H1:DMT-SEVERE_LSC_OVERFLOW.

times. In this case, there is no need to examine a large number of different overlapping overflow flags. However, when new flags are introduced, evaluating their overlaps and the triggers they capture may prove beneficial in understanding their relationships to produce safe, effective vetoes.

5.4 Vetoed and Non-vetoed Triggers

In addition to identifying triggers in specific numbers of overlapping windows, `DQTunePipe` can also identify all triggers vetoed by a particular category (or combination of categories), or conversely, all triggers not vetoed by a specific category (or combination of

triggersWithin_2_overlapFlags_H1.xml

1: ifo	2: startTime	3: endTime	4: snrGreaterThan	5: MainWindowCategories	6: FileGeneratedAt
H1	968457615	968803215	60	1 & 2 & 4 & 5	Fri Nov 18 04:01:45 2011

ClusteredTriggers

1: Count	2: triggerTime	3: snr	4: {DQName; Category(Padding); Original Window Start; Original Window End} etc...	5: Total (Unique, original) windows
1	968707726.710449000	2886.184	Main Window {DMT-OM1_DCPD_OVERFLOW; 2; (0.0 0.0); 968707726.0; 968707729.0} ; {DMT-OM1_OVERFLOW; 2; Window(968707726.0; 968707728.0)}	2
2	968782813.356933000	3669.787	Main Window {DMT-OM1_DCPD_OVERFLOW; 2; (0.0 0.0); 968782813.0; 968782816.0} ; {DMT-OM1_OVERFLOW; 2; Window(968782813.0; 968782815.0)}	2

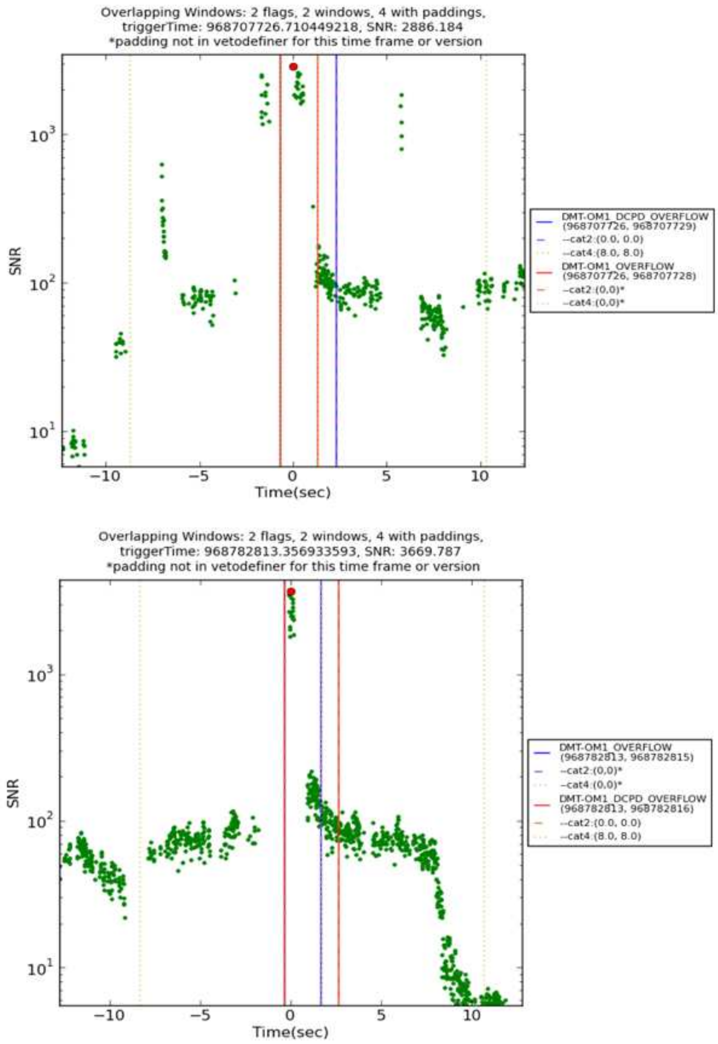


Figure 5.9: List and plots of triggers contained within 2 overlapping windows.

categories). Figure 5.12 shows snippets of two text files produced by the `vetoNonVetoTriggers` task, showing vetoed and non-vetoed triggers in categories 1,2,3,4 and 5. A corresponding lightweight ligo XML file is also produced, but it is better suited for viewing smaller sets of trigger data. The histogram in Figure 5.12 graphically represents the vetoed and non vetoed triggers.

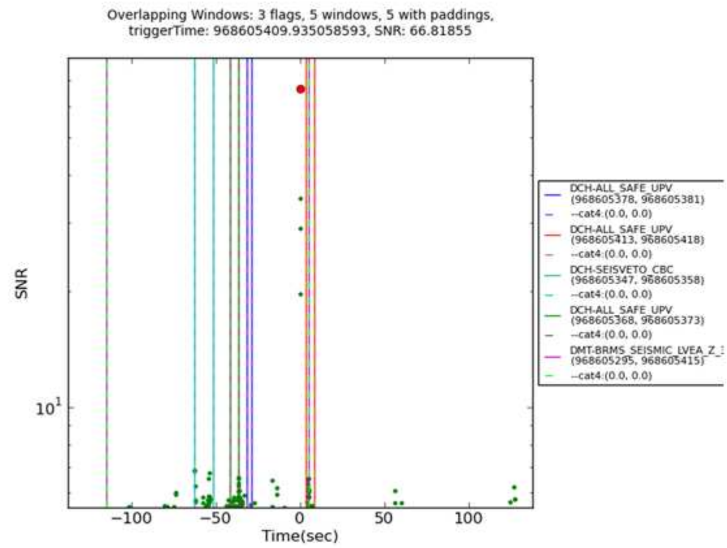
In addition to the histograms, the `vetoNonVetoTriggers` task also invokes the `outliers` task. Figure 5.13 shows an outlier that remains unvetoed if categories 1, 2, 4, and 5 have been applied, but is vetoed by a veto window of category 3. In CBC searches, injections are identified by category 3 vetoes. An injection is seen in the plot in Figure 5.13 at gps time 968654557.931 with SNR of 7.16. The outlier occurs at gps time 968654549.122, 8 seconds before the injection time, and has SNR 121.788. Investigating the outliers of categories 1, 2, 4, and 5, as in Figure 5.13, verifies that category 3 injections are not vetoed by other DQ flags. Investigating outliers can also help LIGO scientists identify significant, unexpected glitches that should be flagged.

triggersWithin_3_overlapFlags_H1.xml

1: ifo	2: startTime	3: endTime	4: snrGreaterThan	5: MainWindowCategories	6: FileGeneratedAt
H1	968457615	968803215	60	1 & 2 & 4 & 5	Fri Nov 18 04:01:45 2011

ClusteredTriggers

1: Count	2: triggerTime	3: snr	4: {DQName; Category(Padding); Original Window Start; Original Window End} etc...	5: Total (Unique, original) windows
1	968605409.935058593	66.81855	Main Window {DMT-BRMS_SEISMIC_LVEA_Z_3_10_HZ_THRESH_2E3; 4; (0.0 0.0); 968605295.0; 968605415.0} ; {DCH-SEISVETO_CBC; 4; (0.0 0.0); Window(968605347.0; 968605358.0)}; {DCH-ALL_SAFE_UPV; 4; (0.0 0.0); Window(968605368.0; 968605373.0)}; {DCH-ALL_SAFE_UPV; 4; (0.0 0.0); Window(968605378.0; 968605381.0)}; {DCH-ALL_SAFE_UPV; 4; (0.0 0.0); Window(968605413.0; 968605418.0)}	5
2	968778029.833984375	4976.165	Main Window {DMT-SEVERE_LSC_OVERFLOW; 2; (0.0 0.0); 968778028.0; 968778034.0} ; {DMT-OM1_DCPD_OVERFLOW; 2; (0.0 0.0); Window(968778030.0; 968778031.0)}; {DMT-OM1_OVERFLOW; 2; Window(968778030.0; 968778032.0)}	3



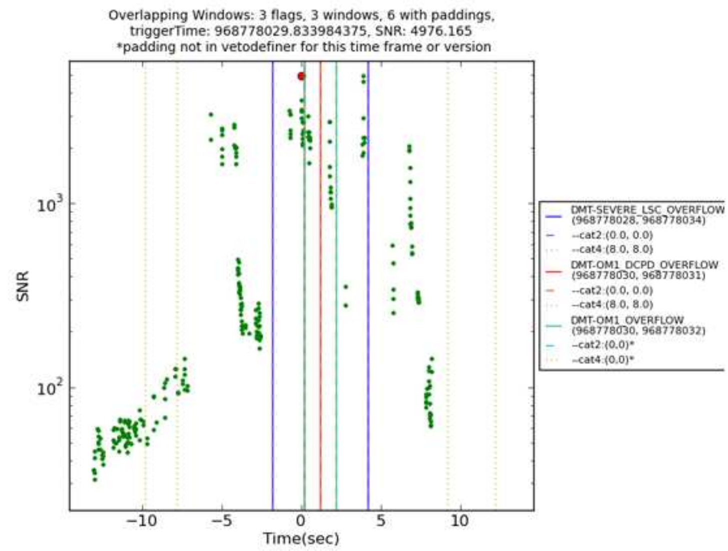


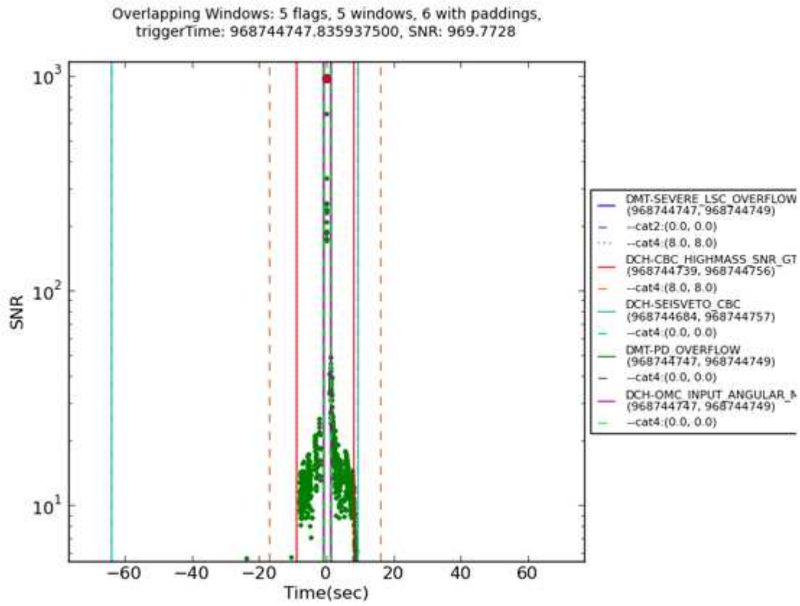
Figure 5.10: List and plots of triggers contained within 3 overlapping flags. The column labeled "Total (Unique, Original)" indicates the number of overlapping DQ windows as originally extracted from the DQ database. The first trigger counted, gps time 968605409.935058593, belongs to three overlapping flags: H1:DMT-BRMS_SEISMIC_LVEA_Z_3_10_HZ_THRESH_2E3, H1:DCH-SEISVETO_CBC, and H1:DCH-ALL_SAFE_UPV. H1:DCH-ALL_SAFE_UPV overlaps with the other two flags through 3 distinct windows. The far right column indicates a total of 5 windows: three from the H1:DCH-ALL_SAFE_UPV flag, and one each from the other two DQ flags.

triggersWithin_5_overlapFlags_H1.xml

1: ifo	2: startTime	3: endTime	4: snrGreaterThan	5: MainWindowCategories	6: FileGeneratedAt
H1	968457615	968803215	60	1 & 2 & 4 & 5	Fri Nov 18 04:01:45 2011

ClusteredTriggers

1: Count	2: triggerTime	3: snr	4: {DQName; Category(Padding); Original Window Start; Original Window End} etc...	5: Total (Unique, original) windows
1	968744747.835937500	969.7728	Main Window {DCH-CBC_HIGHMASS_SNR_GT_250; 4; (8.0 8.0); 968744739.0; 968744756.0} ; {DMT-SEVERE_LSC_OVERFLOW; 2; (0.0 0.0); 4; (8.0 8.0); Window(968744747.0; 968744749.0)}; {DCH-SEISVETO_CBC; 4; (0.0 0.0); Window(968744684.0; 968744757.0)}; {DCH-OMC_INPUT_ANGULAR_MED; 4; (0.0 0.0); Window(968744747.0; 968744749.0)}; {DMT-PD_OVERFLOW; 4; (0.0 0.0); Window(968744747.0; 968744749.0)}	5
2	968773083.105468750	73.37363	Main Window {DCH-ALL_SAFE_UPV; 4; (0.0 0.0); 968773082.0; 968773085.0} ; {DMT-BRMS_SEISMIC_LVEA_Z_3_10_HZ_THRESH_2E3; 4; (0.0 0.0); Window(968772935.0; 968773115.0)}; {DCH-MED_UPCONV_MED; 4; (0.0 0.0); Window(968772986.0; 968773106.0)}; {DCH-SEISVETO_CBC; 4; (0.0 0.0); Window(968773024.0; 968773120.0)}; {DCH-OMC_INPUT_ANGULAR_MED; 4; (0.0 0.0); Window(968773082.0; 968773083.0)}	5



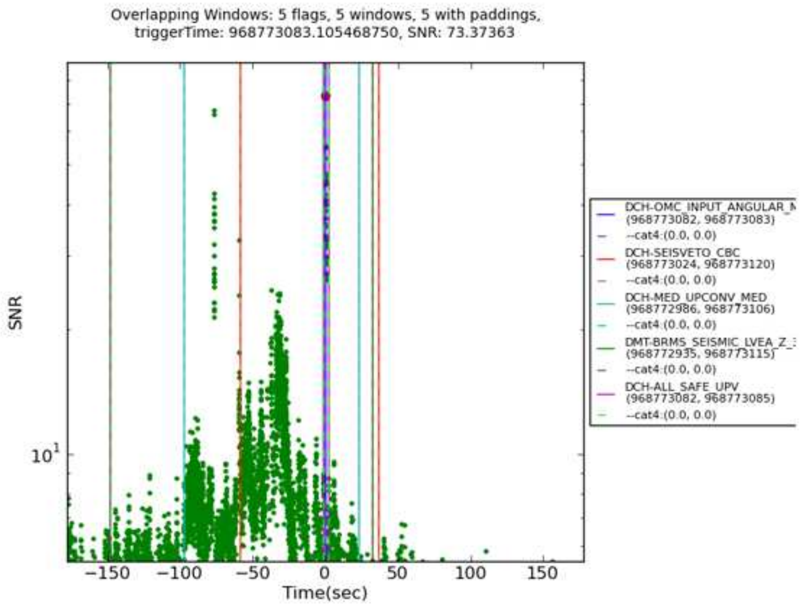


Figure 5.11: List and plots of isolated triggers contained within 5 overlapping flags for the analyzed time. - The plot shows the trigger at gps time 968773083.

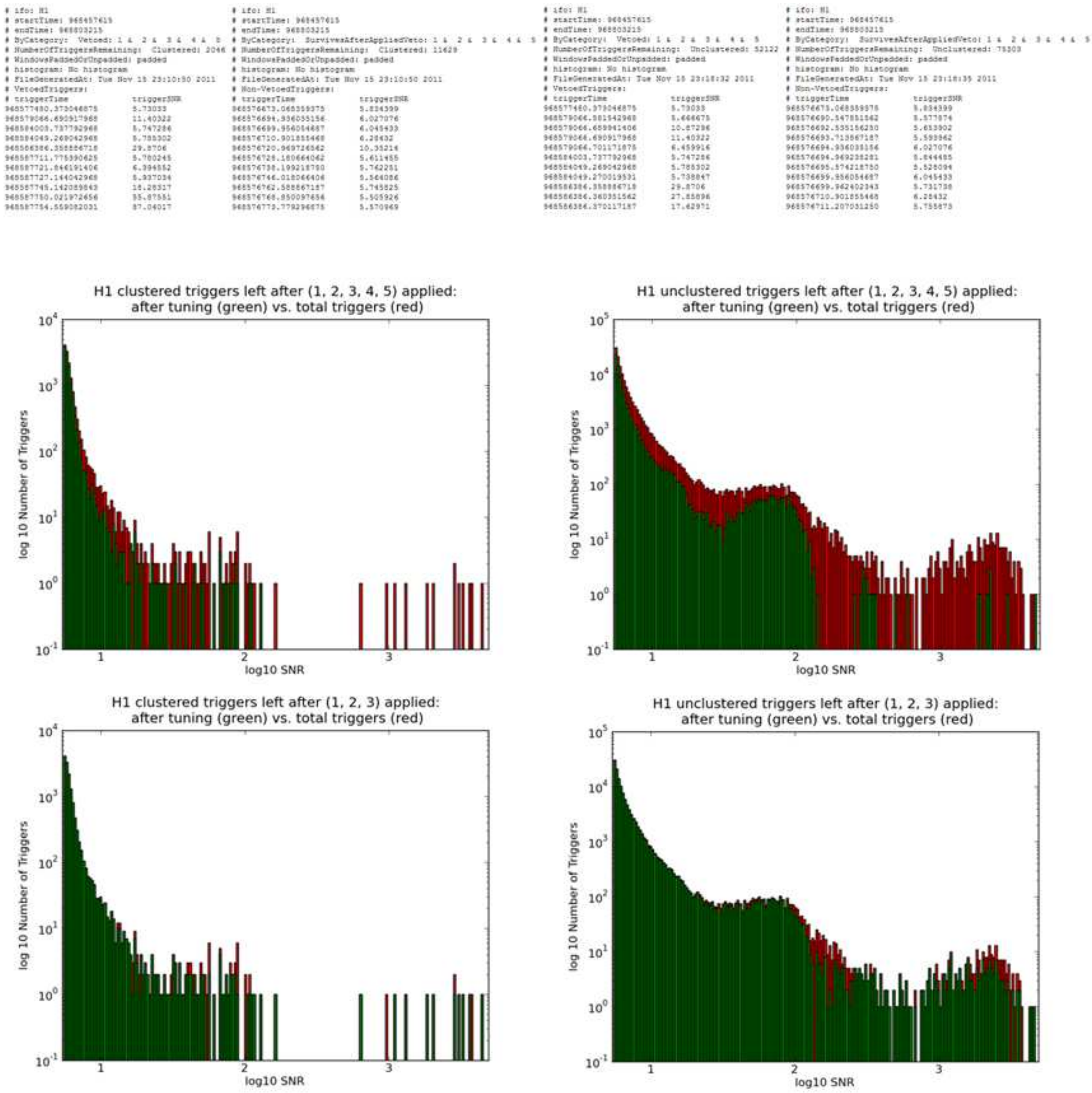


Figure 5.12: Histograms and an excerpt of lists of triggers vetoed and not vetoed by combination of categories - Both clustered and unclustered triggers are shown.

HeaderTable					
1: ifo	2: startTime	3: endTime	4: SurvivesAfterVetoByCategory	5: WindowsPaddedOrUnpadded	6: FileGeneratedAt
L1	968457615	968803215	1 & 2	padded	Wed Nov 16 02:15:59 2011

OutlyingClusteredTriggers					
1: number	2: triggerTime	3: triggerSNR	4: WindowsInSameSegment-Before	5: WindowsInSameSegment-After	6: Nearby Categories
22	968654549.122070000	121.7878	PaddedWindow[DCH-ALL_SAFE_UPV - 4 - [968654187.0 - 968654190.0]]; etc	PaddedWindow[DCH-OMC_INPUT_ANGULAR_MED - 4 - [968654662.0 - 968654664.0]]; etc	2; 3; 4;

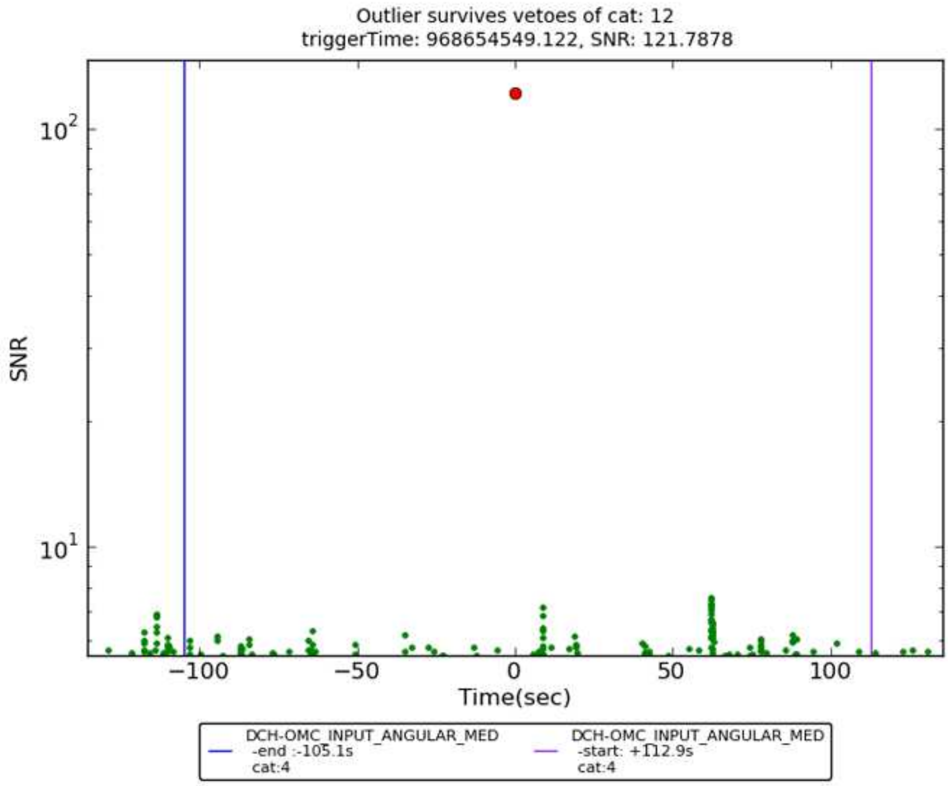


Figure 5.13: An outlier and its nearby windows - An excerpt of the table listing all DQ windows within the same analyzed segment as the outlier. The plot shows the boundaries of the DQ window nearest to the outlier. This outlier, at gps time 968654549.122070312, is found to be vetoed at category 3. It occurs approximately 8 seconds before an injection at gps time 968657557. The full table lists all DQ flags windows in the analyzed segment that includes the outlier.

CHAPTER 6

CONCLUSIONS

The motivation for `DQTunePipe` is to aid LIGO's DetChar team in categorizing and tuning DQ flags. As LIGO scientists gravitate toward the application of more Pythonic analysis tools, it is logical to develop a comprehensive set of Python modules for DetChar use. `DQTunePipe`'s flexibility, versatility, and extensive documentation may make it a useful tool for future LIGO science runs, i.e, Advanced LIGO.

A major purpose for developing `DQTunePipe` was to create a tool in Python for evaluating and tuning DQ flags. The rationale for using Python is that there are many open source and freely available implementations of Python and many existing LIGO tools are written in the Python language. The goal of `DQTunePipe` is a well-documented and easily maintainable tool that allows the user to:

1. Easily run isolated portions of the program;
2. Rerun the program on existing data without repeating the retrieval of all raw data;
3. Allow user-specified configuration parameters;
4. Incorporate new tasks into the program code in a straightforward manner.

`DQTunePipe` achieves these goals through an abstract data layer and modular structure of the program, and by incorporating adaptable configuration parameters.

6.1 Abstraction and Modularity

A major benefit of `DQTunePipe` is the implementation of an object representation of the input data. The `dataSet` object is generated only once (per ifo) when `DQTunePipe` is executed and removes from all tasks any dependence on the format of the input. `DQTunePipe` is structured to allow a developer to add new tasks without modifying the input and likewise modify the input without modifying the tasks.

The modular construction of `DQTunePipe` provides increased flexibility and maintainability. Because of the modular design, future changes to the way that data is processed should require minimal changes in the code. For example, if the format of any of the raw data files were to change, only those functions which process the raw data files would require alteration. The tasks, which rely on the data abstraction layer, would not need to be altered. Since any change is likely to be restricted to individual modules, the software is easier to maintain.

Furthermore, the modularity of each task separates calculations from output. For example, the `FigureOfMerit` task involves two classes, the object `FigureOfMerit` (for generating output) and the object `Metric` (for calculating metric quantities). In other tasks, objects have separate methods to produce output. This is beneficial in that it allows for future modification to the output without affecting the fundamental calculations. For example, it may become preferable to write the metric quantities for each DQ to a database, or store plots as Python objects. Likewise, since each task is represented by an object, the objects can be passed to another task for analysis.

6.2 Configurability and Flexibility

`DQTunePipe` is flexible enough to allow the user a variety of options, yet retains simplicity of use by requiring a minimal configuration file and applying default parameters when not user-provided. The configuration options allow the user to select which tasks to perform and which data subset to evaluate. This makes `DQTunePipe` a good tool for analysis since the user may, for example, choose to execute one task on existing data and repeat calculations on a variety of configurable parameters (i.e. changing the snr threshold for isolated triggers, or applying different padding amounts via a different veto definition file) for comparison.

The use of a configuration file, as well as locally storing the raw data, ensures that `DQTunePipe`'s execution parameters are readily available for repeat analysis. For instance, the user can reuse a configuration file retaining the specifications from a previous run but apply those parameters to a different task by invoking different task option from the command line.

When new DQ flags are proposed, `DQTunePipe`'s `--unpd` option allows users to calculate the metric quantities and outlying triggers based solely on unpadded windows. This is a useful feature when no veto definition file is available.

6.3 Maintenance and Documentation

`DQTunePipe` is also easy to maintain. The structure of `DQTunePipe`'s task are such that input, calculations and output are distinct processes. Therefore, the input source for a particular task can be easily determined, and the output format or naming schema can be changed with no adverse effects on other tasks. For example, `Outliers` uses the `vetoNonVetoedTrigger` object. However, modifications to the displayed output of `vetoNonVetoedTrigger` task have no impact on the `Outliers` task.

The `DataSet` representation and the use of attributes of the `Property` class make it easier to extend `DQTunePipe`'s functionality. It is straightforward to add new tasks using the `DataSet` methods to access the data and the `Property` attributes to access the configurable parameters. Creating an alternative `DataSet` instance using the existing class and its methods could make `DQTunePipe` tasks accessible to non-inspiral gravitational wave searches.

The documentation provided in this thesis describes the steps necessary to incorporate a new task into `DQTunePipe`'s main control structure, including the steps required to add new tasks as user options. The documentation in this thesis also defines the rules for overlapping DQ windows when veto categories for CBC inspiral searches are applied (Figure 3.5.2). The appendices describe the functionality of every module in `DQTunePipe`, and those descriptions are likewise included in the code itself.

6.4 Summary

`DQTunePipe` accomplishes its goals and the requirements outlined in Chapter 3 via an abstract data layer, modular construction, and flexible configuration options. By satisfying its objectives, along with the inclusion of comprehensive documentation, `DQTunePipe` surpasses the previously used collection of `MATLAB` scripts in documentation, versatility, usability, and simplicity of use.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Location of vetodefiner files: https://www.lsc-group.phys.uwm.edu/ligovirgo/cbc/public/segments/s6/. *LIGO-internal*, 2011. 17, 29

[2] Bruce Allen, Warren G. Anderson, Patrick R. Brady, Duncan A. Brown, and Jolien D. E. Creighton. FINDCHIRP: an algorithm for detection of gravitational waves from inspiraling compact binaries. *General Relativity and Quantum Cosmology (gr-qc), arXiv:gr-qc/0509116v1*, 2005. 11

[3] Duncan Brown, Larne Pekowsky, and Ping Wei. https://www.lsc-group.phys.uwm.edu/daswg/projects/glue/doc/ and https://www.lsc-group.phys.uwm.edu/daswg/projects/glue/epydoc/bin/ligolw_segment_query, 2011. 23

[4] Duncan A. Brown. Searching for Gravitational Radiation from Binary Black Hole MACHOS in the Galactic Halo. *PhD Thesis, University of Wisconsin-Milwaukee, gr-qc, arXiv:0705.1514v1*, 2004. 8, 10, 17

[5] Marco Cavaglià. https://wiki.ligo.org/cbcdataqualityveto/s6highmassvetoes. *LIGO-internal*, 2010. 17

[6] Nelson Christensen. Veto studies for LIGO inspiral triggers. *Classical and Quantum Gravity, 22, S1059-S1068*, 2005. 10, 14, 17

[7] Nelson Christensen. https://wiki.ligo.org/detchar/s6dqflags, https://virgo.physics.carleton.edu/trac/upv-trac. *LIGO-internal*, 2010. 15

[8] Nelson Christensen. LIGO S6 Detector Characterization Studies. 2010. 14, 16

[9] The LIGO Scientific Collaboration. Tuning Matched Filter searches for Compact Binary Coalescence. *LIGO-technical note, T070109-01*, 2007. 11

[10] J. Slutsky *et al.* Methods for Reducing False Alarms in Searches for Compact Binary Coalescences in LIGO Data. *Class. Quantum Grav. 27, arXiv:1004.0998*, 2010. 10, 14, 15, 17, 18, 20

[11] L. Blackburn *et al.* The LSC Glitch Group : Monitoring Noise Transients during the fifth LIGO Science Run. *Class.Quant.Grav.25:184004, LIGO-P080016-01, arXiv:0804.0800v2 [gr-qc]*, 2008. 13, 15

[12] Peter Fritschel, Rolf Bork, Gabriela González, Nergis Mavalvala, Dale Ouimette, Haisheng Rong, Daniel Sigg, and Michael Zucker. Readout and control of a power-recycled interferometric gravitational wave antenna. *LIGO-P000008-A-D*, 2000. 8

[13] Evan Goetz and Keith Riles. PlaneMon: Airplane Detection Monitor. *LIGO DCC:T050174-00-D*. 15

[14] Lisa Marie Goggin. A Search for Gravitational Waves from Perturbed Black Hole Ringdowns in LIGO Data. *PhD Thesis, California Institute of Technology, gr-qc, arXiv:0908.2085v1*, 2009. 6, 8, 10

[15] Gabriela González. https://wiki.ligo.org/detchar/dmtflags#dmt_flags. *LIGO-internal*, 2010. 15, 17

[16] James B. Hartle. Gravity, an Introduction to Einstein's General Relativity. *Addison Wesley Publishing*, 2003. 2, 5

[17] R. A. Hulse and J. H. Taylor. Discovery of a pulsar in a binary system. *Astrophysical Journal, vol. 195, Jan. 15, 1975, pt. 2, p. L51-L53*, 1975. 6

[18] Tomoki Isogai, the LIGO Scientific Collaboration, and the Virgo Collaboration. Used percentage veto for LIGO and Virgo binary inspiral searches. *J. Phys.: Conf. Ser. 243 012005 doi: 10.1088/1742-6596/243/1/012005*, 2010. 15

[19] Andres Rodriguez. Reducing False Alarms in Searches for Gravitational Waves from Coalescing Binary Systems. *M.S. Thesis, Louisiana State University, gr-qc, arXiv:0802.1376v1*, 2007. 11

[20] Peter R Saulson. Fundamentals of Interferometric Gravitational Wave Detectors. *World Scientific*, 1994. 7

[21] Joshua Smith. https://wiki.ligo.org/detchar/dchflags. *LIGO-internal*, 2011. 16

LIST OF APPENDICES

APPENDIX A

APPENDIX A

MAIN CONTROL STRUCTURE OF THE `DQTUNEPIPE`

A.1 `DQTunePipe.py`

DQTunePipe is the main control structure which is used to execute tasks and is initialized via the command line, by: ***`python DQTunePipe -f <configuration File>`*** *`<ADDITIONAL ARGUMENTS>`*:

Acquires Raw Data and process Intermediate (Trigger) Data, as necessary.

For each `ifo`:

Establishes the `dataSet` object.

If requested by user (not executed by default), executes the deadtime task:

Instantiates the `WriteDeadTime` object and generates the DeadTime Table for `Padded` and `Unpadded Windows`

Executes the Figures of Merit task:

Instantiates the `FiguresOfMerit` objects for both `Padded` and `Unpadded Windows` with both `Clustered` and `Unclustered Triggers`,

Calls the `writeSummary` and `writeMetricTable` methods for those objects.

Executes the Isolated and Overlapping Windows and Triggers task:

Instantiates the `OverlappingPaddedWindows` object. Then for both isolated and overlapping windows and both clustered and unclustered `Triggers`:

Calls the `writeOverlappingWindows` and `writeOverlappingTriggers` methods,

Calls the `createTriggerDictionary` method of `OverlappingWindows` to instantiate a `PlotTriggers` object. Isolated (Overlapping) Triggers are plotted against a background of `Unclustered Triggers`.

Executes the Vetoed, Non-Vetoed, and Outlying Triggers task:

Instantiates a `VetoedAndNonVetoedTriggers` object for both `Padded` and `Unpadded Windows` and for both clustered and unclustered `Triggers`:,

Calls the `writeFiles` method of the `VetoedAndNonVetoedTriggers` object.

The `Outliers` object is instantiated when the `writeFiles` method is called, as is the `histogram` method of the `VetoedAndNonVetoedTriggers` object.

Writes the output of tasks to user-specified output directory: Additionally, a log file, ***dqtunepipe.log***, containing information and status of the run, as well as any errors, is written to the directory from which the user executes the `DQTunePipe`.

APPENDIX B

APPENDIX B

PYTHON DATA QUALITY TUNER OBJECTS - SET UP

B.1 `properties.py`

B.1.1 Allowed Values

Lists of currently supported values used throughout the ***`Properties`*** *class.*

`AllowedIFOS = ['H1','L1','H2','V1']` List of ifo values currently supported. `DQTunePipe` may be expanded to support additional ifo values.

`supported_gw_programs = ['INSPIRAL','RINGDOWN']` Search programs currently supported. `DQTunePipe` may be expanded to support additional GW Programs.

`supported_inj_cats = [1,2,3,4,5]` Identifies the category values allowed for injections.

B.1.2 class `PropertyError`

`PropertyError(Exception)` *The PropertyError class is used to raise exceptions in the* ***`Properties`*** *class.*

B.1.3 class `Properties`

`Properties(object)` *The attributes of the* ***`Properties`*** *class are used to identify the properties used by* ***`DQTunePipe`****, and are referenced by the* ***`DQTunePipe`*** *via* ***`properties.attribute`****.*

Attributes that MUST be specified by user, i.e. no defaults: these attributes have `_get_` and `_set_` methods.

CONFIGURATION_FILENAME Identifies the configuration file to be used. Command-line argument only.

INITIAL_START Identifies the starting time of data to be used.

INITIAL_END Identifies the ending time of data to be used.

OUTPUT_DIRECTORY Identifies the directory to in which to write output data.

VETODEFINERFILE_FULLPATH Identifies the full path location of the *vetodefiner* file.

TRIGGER_SOURCE_DIRECTORIES Identifies the initial directories of trigger input files.

IHOPE_TRIGGER_SOURCE Identifies the initial directory location of *ihope* trigger input files.

CLUSTERED_FORMAT Identifies the nomenclature used in clustered data input files.

UNCLUSTERED_FORMAT Identifies the nomenclature used in unclustered data input files.

DQ_SCIENCE_H1 Identifies the DQ *science* flag associated with the *H1* interferometer.

DQ_SCIENCE_L1 Identifies the DQ *science* flag associated with the *L1* interferometer.

Attributes that MAY be specified by user, i.e. have defaults, and are available for configuration from the command line: these attributes have `_get_`, `_set_`, and `_get_default_` methods.

IFOS Identifies the interferometers.

THRESHOLDS Identifies the snr thresholds to be used in calculating the metric quantities.

PADDINGS Identifies whether padded or unpadded data should be used.

IS_CLUSTERED Identifies whether `DQTunePipe` should use *clustered* or *unclustered* trigger data.

DEBUG Identifies whether `DQTunepipe` writes additional debugging information to log.

PRINT_VALUES Identifies whether the `DQTunepipe` writes the values of the `properties` attributes to console and then exits.

DO_FOM_TABLE Identifies whether `DQTunePipe` should execute the tasks for `figureOfMerit`.

DO_DEAD_TIME_TABLE Identifies whether `DQTunePipe` should execute the task for calculating just the `deadTime`.

DO_ISOLATED Identifies whether `DQTunePipe` should execute the tasks for *isolated* triggers.

DO_ISOLATED_PLOTS Identifies whether `DQTunePipe` should execute the tasks for *isolated* trigger plots.

DO_OVERLAPPING Identifies whether `DQTunePipe` should execute the tasks for *overlapping* triggers.

DO_OVERLAPPING_PLOTS Identifies whether `DQTunePipe` should execute the tasks for *overlapping* triggers plots.

DO_OUTLIERS Identifies whether `DQTunePipe` should execute the tasks for `Outliers`, including `plotOutlyingTriggers`.

DO_HISTOGRAM Identifies whether `DQTunePipe` should execute the task for `histogram` in `VetoedAndNonvetoedTriggers`.

DO_VETOED_NONVETOED_TRIGGERS Identifies whether `DQTunePipe` should execute the task for `VetoedAndNonvetoedTriggers`.

Attributes that MAY be specified by user, i.e. have defaults, but are only available to be configured from the configuration file: these attributes have `_get_`, `_set_`, and `_get_default_` methods.

SNR_ISOLATED_THRESH Identifies the snr threshold to be used in determining the isolated and overlapping triggers.

OUTLIER_THRESH Identifies the snr threshold to be used in determining the outlier triggers.

MAX_OVERLAPS Identifies the maximum number of overlapping windows for a given `DQ` that the `DQTunePipe` writes to file.

MIN_OVERLAPS Identifies the minimum number of overlapping windows for a given `DQ` that the `DQTunePipe` writes to file.

DQ_XML_DIRECTORY Identifies the location of the directory storing the raw DQ xml files.

DQ_SERVER_LOCATION Identifies the server location of the DQ database.

LIGO_LW_XSL Identifies the location of style-sheet to display output XML files, *mod_ligolw.xsl*.

UNCLUST_TRIGGER_FILE_H1, **UNCLUST_TRIGGER_FILE_H2**, **UNCLUST_TRIGGER_FILE_L1**, and **UNCLUST_TRIGGER_FILE_V1** Identifies the *unclustered* trigger text file to use for *H1*, *H2*, *L1*, and *V1*.

CLUST_TRIGGER_FILE_H1, **CLUST_TRIGGER_FILE_H2**, **CLUST_TRIGGER_FILE_L1**, and **CLUST_TRIGGER_FILE_V1** Identifies the *clustered* trigger text file to use for *H1*, *H2*, *L1*, and *V1*.

ANALYZED_TRIGGER_SEGMENT_FILE_H1, **ANALYZED_TRIGGER_SEGMENT_FILE_H2**, **ANALYZED_TRIGGER_SEGMENT_FILE_L1**, and **ANALYZED_TRIGGER_SEGMENT_FILE_V1** Identifies the analyzed segment trigger text file to use for *H1*, *H2*, *L1*, and *V1*.

GW_PROGRAM Identifies the gravitational wave search data to use. Currently `DQTunePipe` supports only trigger data from *INSPIRAL* and *RINGDOWN* searches.

INJ_CAT Identifies the category assigned to vetoes from hardware injections.

Attributes that may NOT be specified by user, and their values: (includes filenames, directory names, internal-only references, etc): these attributes have only _get_ methods.

H1 Identifies the *H1* interferometer.

H2 Identifies the *H2* interferometer.

L1 Identifies the *L1* interferometer.

V1 Identifies the *V1* interferometer.

DATA_DIRECTORY Identifies the directory to in which to store input data: OUTPUT_DIRECTORY*/data/*

DATA_SUMMARY_DIRECTORY Identifies the location of the *summary* directory that stores intermediate data: DATA_DIRECTORY*/summary/*

DQ_LIST_FILE Identifies the location of the file to contain the list of DQ flag names: DATA_DIRECTORY*/DQ_LIST_INITIAL*

VETO_DEFINER_FILE Identifies the location of the copy of the *vetodefiner* file: DATA_DIRECTORY*/vetodefiner.xml*

TRIGGER_FILES Identifies the trigger directory locations of the copied trigger data associated with each interferometer: DATA_DIRECTORY*/triggers-ifo/*

IFO_MAIN_OUTPUT Identifies the main output directory for each interferometer: OUTPUT_DIRECTORY*/ifo*_INITIAL_START-INITIAL_END/

FOM_OUTPUT Identifies the directory for figuresOfMerit task output for each interferometer: IFO_MAIN_OUTPUT[ifo]*/FiguresOfMerit/*

FOM_SUMMARY_OUTPUT Identifies the subdirectory for figuresOfMerit task summary output: *FOM_DQSummaries/*, in each IFO_MAIN_OUTPUT[ifo]

CLUSTERED_SUBDIRECTORY Identifies the output subdirectories for each task executing on clustered triggers: *clustered_triggers*

UNCLUSTERED_SUBDIRECTORY Identifies the output subdirectories for each task executing on unclustered triggers: *unclustered_triggers*

ISOLATED_OVERLAPPING Identifies the subdirectory for isolated and overlapping DQ windows and triggers for each interferometer: IFO_MAIN_OUTPUT[ifo]*/IsolatedOverlappingWindowsAndTriggers/*

ISOLATED_OVERLAPPING_WINDOWS Identifies the subdirectory for isolated and overlapping DQ windows: *isolatedOverlappingWindows/*

ISOLATED_OVERLAPPING_TRIGGERS Identifies the subdirectory for isolated and overlapping triggers: *isolatedOverlappingTriggers/*

ISOLATED_OVERLAPPING_PLOTS Identifies the subdirectory for plots of isolated and overlapping triggers: *isolatedOverlappingPlots/*

VETOED_NONVETOED_TRIGGERS Identifies the subdirectory for vetoed and non vetoed triggers: *vetoedAndNonVetoedTriggers/*

OUTLIERS Identifies the subdirectory for outlier triggers: *outliers/*

OUTLIER_PLOTS Identifies the subdirectory for outlier plots: *outlierPlots/*

HISTOGRAMS Identifies the subdirectory for histograms: *histograms/*

DQ_SCIENCE_FLAGS Internally assigns DQ_SCIENCE_H1 and DQ_SCIENCE_L1 to DQ_SCIENCE_FLAGS[ifo] for use throughout DQTunePipe.

UNCLUST_TRIGGER_FILE Internally assigns UNCLUST_TRIGGER_FILE_H1 and UNCLUST_TRIGGER_FILE_L1 to UNCLUST_TRIGGER_FILE_FLAGS[ifo] for use throughout DQTunePipe.

CLUST_TRIGGER_FILE Internally assigns CLUST_TRIGGER_FILE_H1 and CLUST_TRIGGER_FILE_L1 to CLUST_TRIGGER_FILE_FLAGS[ifo] for use throughout DQTunePipe.

ANALYZED_TRIGGER_SEGMENT_FILE Internally assigns ANALYZED_TRIGGER_SEGMENT_FILE_H1 and ANALYZED_TRIGGER_SEGMENT_FILE_L1 to ANALYZED_TRIGGER_SEGMENT_FILE_FLAGS[ifo] for use throughout DQTunePipe.

Additional methods available to the Properties object.

set_defaults Method to assign all default values to those properties which do not yet have a value assigned.

show_all_set Method to display values assigned to configurable properties.

show_all Method to display all values assigned properties.

add_slash Method used to ensure that user-provided directory names contain an ending '/'.

APPENDIX C

APPENDIX C

PYTHON DATA QUALITY TUNER OBJECTS - TASKS AND DATA

C.1 `entities.py`

C.1.1 class `DataSet`

`DataSet(object)` *A dataset object represents all information necessary to run the `DQTunePipe`. Initialized with `DataSet(ifo, start, end, scienceFlagName)`.*

Each of the following is an attribute of DataSet, with associated *get* and *set* methods:

`ifo` The interferometer of interest for this `DataSet`.

`start` The gps start time of the period of interest for this `DataSet`.

`end` The gps start time of the period of interest for this `DataSet`.

`analyzedSegments` The list of analyzed segments for this `DataSet`.

`clusteredTriggers` The list of clustered triggers for this `DataSet`.

`unclusteredTriggers` The list of unclustered triggers for this `DataSet`.

`threshold` The list of snr thresholds of interest for this `DataSet`.

`dqs` The list of **`DQ`** flag objects in this `DataSet`.

`paddings` The list of the **`Padding`** objects in this `DataSet`.

The following methods are part of the class `DataSet`:

getScience Returns the science data, i.e. the **DQ** flag object identified by the scienceFlagName in the initialization of this DataSet.

getTimeAnalyzed Returns the sum of the duration of all of the windows of the science data.

getDQByName(dqname) Returns an individual **DQ** flag object identified by its dqname.

getPaddingsByDQ(dq) Returns **padding** objects associated with an individual dq (**DQ** object) identified by its dqname.

getPaddingByDQCategory(dq, category) Returns **padding** objects associated with an individual dq (**DQ** object) identified by its dqname and category.

getUndefinedDQs Returns the names of all DQ flags that have paddings but do not have associated **DQ** objects in this DataSet.

C.1.2 class Padding

Padding(object) *A Padding object represents the lengths of time (in seconds), per applicable category, to be added to the beginning and end of a specific DQ Window object. Initialized with Padding(dqName, category, left, right, paddedSegmentWindow).*

Each of the following is an attribute of Padding, each with associated *get* and *set* methods:

dqname The name of the DQ flag to which this padding applies.

category The category to which this padding applies.

left The length of time to be applied before the start of a window.

right The length of time to be applied after the end of a window.

paddedSegmentWindow The segment of time to which this padding applies.

C.1.3 class `DQ`

`DQ(object)` *A `DQ` object has a name and windows, and represents a DQ flag. Initialized with `DQ(dataSet, name)`.*

Each of the following is an attribute of `DQ`, each with associated *get* and *set* methods:

`dataSet` The **`DataSet`** to which this `DQ` belongs.

`name` The name of this `DQ` flag.

`windows` The list of windows associated with the `DQ`.

The following items are methods of the class `DQ`:

`intersectWindows(otherWindows, category=None)` Returns a set of `resultWindows` that are the intersection of the *self* windows of this `DQ` and the set of *other* windows that `intersectWindows` requires as an argument. May also take a category value as an argument to return a set of `resultWindows` that are the intersection of *self*.**`PaddedWindows`** and *other* windows.

`getNumOfWindowsThatIntersect(otherWindows, category=None)` Returns the number of windows in *self* `DQ` that intersect *otherWindows*. Requires *otherWindows* as an argument and may take a category value as an argument. If a category argument is given, compares *self*.**`PaddedWindows`** for that category to *otherWindows*. (Uses `intersectWindows`).

`getMedianDurationOfWindowsThatIntersect(otherWindows, category=None)` Returns the median duration of `resultWindows` from `intersectWindows`. Requires *otherWindows* as an argument and may take a category value as an argument. If a category argument is given, compares *self*.**`PaddedWindows`** for that category to *otherWindows*. (Uses `intersectWindows`).

getScienceDuration(category=None) Returns the sum of the duration of all of the resultWindows from the intersection of the *self* windows and the windows of science data. May accept a category value as an argument. If a category argument is given, compares *self*.**PaddedWindows** for that category to *otherWindows*.

getCategories The list of veto categories, as defined by paddings, which may be defined from this DQ flag.

getPaddings The list of paddings associated with this DQ flag.

getPaddingByCategory(category) Returns padding used for specific category applicable to this DQ. Requires a category value argument.

getPaddedWindows(category) Return a list of windows that have been padded according to this dq, category, version and time frame (paddedSegmentWindow) of applicable padding. Combines windows (of same DQ) that overlap due to padding into single window. Requires a category value argument.

getUnpaddedWindows Returns a list of **UnpaddedWindows** for this DQ.

getDeadTimeCalc(category=None) Returns the deadtime of the DQ, the percentage of time flagged by this DQ while science data was collected *self*.getScienceDuration per the total science time, dataSet.getTimeAnalyzed.

C.1.4 class Window

Window(object) *A window object represents a period of time. It is initialized with Window(start, end, version=None), (version is an optional initialization argument, as it is only applicable to DQ windows).*

Each of the following is an attribute of Window with associated *get* and *set*

methods:

start Start time of window.

end End time of window.

version The version number of the window, for `DQ` flag windows.

The following items are methods of the class `Window`:

getDuration Returns duration of window.

overlaps(other) Takes *other* window as argument. Returns `True` if *self* overlaps in time with *other* window (exclusive of start/end times), else returns `False`.

adjacent(other) Takes *other* window as argument. Returns `True` if *self* is adjacent in time with *other* window at start or end times, else returns `False`.

containedBy(other) Takes *other* window as argument. Returns `True` if *self* is contained in time within *other* window, else returns `False`.

contains(trigger) Takes *trigger* as argument. Returns `True` if *trigger* is contained in time within *self* window, else returns `False`.

C.1.5 class `AnalyzedSegmentWindow`

AnalyzedSegmentWindow(Window) *A window representing a full segment. Initialized with `AnalyzedSegmentWindow(start, end)`.*

Each of the following items is an attribute of the class `AnalyzedSegmentWindow`, each with associated *get* and *set* methods:

getStart Returns the `start` of this window.

getEnd Returns the `end` of this window.

The following items are methods of the class `Window`:

getDuration Returns duration of window.

C.1.6 class `PaddedWindow`

`PaddedWindow(Window)` *A window object augmented with DQ and category information. Initialized with `PaddedWindow(window, dq, category)`.*

Each of the following is an attribute of `PaddedWindow` with associated *get* and *set* methods:

- **`dq`** The **`DQ`** object to which this window belongs.
- **`category`** The category to which this window will be assigned.

The following items are methods of the class `PaddedWindow`:

- **`getPadding`** Returns the **`Padding`** which will be assigned to this window based on the **`DQ`** and category.
- **`getStart`** Returns the `start` of this window adjusted by *self*.`padding.left`.
- **`getEnd`** Returns the `end` of this window adjusted by *self*.`padding.right`.
- **`getWindow`** Returns the base `Window`.

C.1.7 class `UnpaddedWindow`

`UnpaddedWindow(Window)` *A window with DQ information. Initialized with `UnpaddedWindow(window, dq)`.*

Each of the following is an attribute of `UnpaddedWindow`, each with associated *get* and *set* methods:

- **`dq`** The **`DQ`** to which this window belongs.

The following items are methods of the class `UnpaddedWindow`:

- **`getStart`** Returns the `start` of this window.
- **`getEnd`** Returns the `end` of this window.
- **`getWindow`** Returns the base `Window`.

C.1.8 class `Trigger`

`Trigger(object)` *A trigger object represents a trigger by a point in time, and an SNR value. Initialized with* ***`Trigger(time, snr, mass=None, timeString=None)`****, (mass and timeString are optional initialization arguments; mass is not used by* ***`DQTunePipe`****).*

Each of the following is an attribute of `Trigger` with associated *get* and *set* methods:

`time` Chirp time of `trigger`.

`snr` Signal-To-Noise ratio of `trigger`.

`mass` Template mass of `trigger`.

`timeString` String representation of `time`.

The following items are methods of the class `UnpaddedWindow`:

`getCoarseTime` Returns *self*.`time` rounded to the nearest integer.

`containedByWindow` Check if `time` is within a specific window. Takes a window as argument, returns `True` if *self*.`time` is between window.`start` and window.`end`, else returns `False`.

C.1.9 `def mergeWindows`

`mergeWindows` *Stand alone method called by* ***`mergeWindows(windows, adjacent_parameter=False)`****.*

`mergeWindows` Method for merging windows that overlap, requires a sorted list of `Windows` as argument and may take an additional argument `adjacent_parameter`. If `adjacent_parameter`=`True`, will merge windows that are immediately adjacent, i.e. end time of one window equals start time of next window, and if `adjacent_parameter`=`False` (default), will not merge windows that are immediately adjacent. Returns a list of `Window` objects.

C.2 `figuresOfMerit.py` and `metric.py`

C.2.1 class `FiguresOfMerit`

`FiguresOfMerit(object)` *The FigureOfMerit class is used to create the output xml files containing the calculated veto metric quantities. Initialized by* ***`FiguresOfMerit(dataSet, thresholds, isPadded, isClustered)`****.*

Each of the following is an attribute of `FiguresOfMerit`:

- **`dataSet`** The **`DataSet`** to which this `FigureOfMerit` belongs.
- **`thresholds`** The list of snr thresholds to be used in calculating metric quantities.
- **`isPadded`** `True` (`False`) value indicating whether `PaddedWindows` (`UnpaddedWindows`) are to be used in calculating metric quantities.
- **`isClustered`** `True` (`False`) value indicating whether *clustered* (*unclustered*) triggers are to be used in calculating metric quantities.
- **`triggersAboveThresh`** Dictionary of *clustered* (*unclustered*) triggers, where each key is an `snr` threshold, and the corresponding value is a list of triggers above that `snr` threshold.
- **`metrics`** The list of metric quantities calculated via `calculateMetrics`, for each threshold with the corresponding set of triggers.

The following are methods of the class `FiguresOfMerit`:

- **`getDataSet`** Returns `dataSet`.
- **`getMetrics`** Returns `metrics`.
- **`getThresholds`** Returns `thresholds`.
- **`checkIfPadded`** Returns `isPadded` value.
- **`checkIfClustered`** Returns `isClustered` value.

getTriggersAboveThresh For each snr `threshold` in list of `thresholds`, returns the set of *clustered* (*unclustered*) triggers with snr above that `threshold` in `dataSet`, according to the value of `checkIfClustered`.

getNumberOfTriggersAboveThresh(threshold) Returns the number of triggers with snr above `threshold`.

calculateMetrics(threshold, triggers) Returns a list of `Metric` objects (of all padded (unpadded) `DQ` objects), calculated with triggers with snr above a given `threshold` value. Uses `PaddedWindows` (`UnpaddedWindows`) in `dataSet`, according to the value of `checkIfPadded`.

calculateChiMaxForMetrics(metrics) For each **`DQ`** in **`metrics`**, returns the maximum value of `chiSquared` (across all `chiSquared` values for that **`DQ`**) and the corresponding snr `threshold` value.

writeMetricTable(ifo) Creates the FiguresOfMerit directory structure (with *clustered* (*unclustered*) subdirectories) to house the figure of merit data. Copies stylesheet from `properties.LIGO_LW_XSL` into subdirectories. Uses `writeMetricData` to write the `DQ` object and its `metric` quantities for each snr `threshold` into an xml in *table/colummn/stream* format to be interpreted by stylesheet.

writeMetricData(ifo, threshold, dq, doc, streamTag) Writes the `metric` quantities for each `dq` (`DQ` object), for each snr `threshold` into the *stream* identified by the `streamTag` of the xml file identified as `doc`.

writeSummary(ifo) Creates figures of merit summary subdirectory in *clustered* (*unclustered*) subdirectory. Loops over `DQ` objects, per `Padding`, to write a txt file using `writeSummaryFile` that contains `DQ` object and `metric` details.

writeSummaryFile(file, metrics, ifo, dq, category) Writes details of `dq` (`DQ` object) and its associated `metric` quantities to a txt file identified as `file`, for each `Padding` applied to that `dq`.

C.2.2 class `Metric`

`Metric(object)` *The Metric class is used to determine metric quantities for a specific DQ object and category, based on a set of triggers with snr values above a given threshold. Initialized with* `Metric(dq, triggers, threshold, figureOfMerit, category)` *(category may be None).*

Each of the following is an attribute of `Metric`:

`dq` The specific `DQ` object to which this `Metric` refers.

`triggers` The set of triggers on which this `Metric` refers.

`threshold` The snr `threshold` on which this `Metric` is based.

`category` The veto category determining the padding amounts added to windows of the specific `DQ` object to which this `Metric` refers.

`figureOfMerit` The `FigureOfMerit` object to which this `Metric` refers.

`chiSquared` The χ^2 value for this `Metric`, i.e. the sum of the terms in equation 2.4, across all windows for the `dq`.

`chiDegFreedom` Degrees of freedom in χ^2, i.e. a count of the number of terms in the sum of χ^2 (Equation 2.4) less 1.

`chiSquaredRatio` The ratio of `chiSquared` per the inverse cumulative distribution function (confidence of 95%).

`usedWindows` The number of windows of `dq` that contain triggers.

`totalTriggersVetoed` The number of triggers contained by windows of `dq`.

`effiency` The *efficiency* of the DQ flag, i.e. the percentage of `totalTriggersVetoed` per the number of triggers above the snr `threshold` via `figureOfMerit.getNumberOfTriggersAboveThresh(threshold)`.

usePercent The *use percentage* of the DQ flag, i.e. the percentage of usedWindows per number of windows of `dq` that intersect with windows of `science` data (via `dq.intersectWindows(dq.dataSet.science.windows, category)`).

corrTerm The correlation term relating *effiency, usePercent,* and *deadtime*; `effiency`/(`usePercent` ∗ `dq.getDeadTimeCalc(category)`).

C.3 `writeDeadTime.py`

C.3.1 class `WriteDeadTime`

WriteDeadTime(object) *The WriteDeadTime class is used to write the deadTime of each DQ to an xml file. Initialized by* `WriteDeadTime(dataSet, isPadded)`.

Attributes of `WriteDeadTime`:

dataSet The **DataSet** to which this `WriteDeadTime` belongs.

isPadded `True`(`False`) value indicating whether `PaddedWindows` (`UnpaddedWindows`) are to be used in calculating deadTime.

The following are methods of the class `WriteDeadTime`:

getDataSet Returns `dataSet`.

checkIfPadded Returns `isPadded` value.

writeDeadTimeXML(ifo) Writes the xml containing the deadTimes calculated for each `DQ` in `DQDataSet` Object in table/columnn/stream format to be interpreted by the stylesheet, `properties.LIGO_LW_XSL`. Uses `writeDeadTimeData` to write the DQ object and its deadTime for padded (unpadded) windows.

writeDeadTimeData(ifo, dq, doc, streamTag) Write the deadTime info for each DQ.

C.4 `OverlappingWindows.py` and `plotTriggers.py`

C.4.1 class `OverlappingWindows`

`OverlappingWindows(object)` *The OverlappingWindows class is used to determine and store which `DQ` windows overlap with other `DQ` windows. Initialized by `OverlappingWindows(dataSet)`.*

Each of the following is an attribute of `OverlappingWindows`:

- **`dataSet`** The **`DataSet`** to which this `FigureOfMerit` belongs.
- **`overlappingWindows`** A python `dictionary` where the *`key`* is the `window` object of interest and the *`value`* is a list of other `window` objects that overlap in time with the (*`key`*) `window` object of interest.

the following are methods of the class `OverlappingWindows`:

- **`getOverlappingWindows`** Returns `overlappingWindows`.
- **`addOverlappingWindows(window, overlappingwindow)`** Adds an `overlappingwindow` to `overlappingWindows[window]`.
- **`getTriggersAboveIsolatedThresh(isClustered)`** Returns clustered (or unclustered) triggers with snr values above above a given threshold.
- **`makeOverlappingWindows`** Loops over all `DQ` objects with assigned categories in `dataSet` and compares the `PaddedWindow` object each `DQ` to the *other* `PaddedWindow` objects every *other* `DQ`. If the *other* `PaddedWindow` overlaps in time with the first `Paddedwindow`, per the category rules described in section 3.5.2, then calls *self.*`addOverlappingWindows` to add the *other* window to the list of `overlappingWindows` *keyed* by the first `Paddedwindow`. *Note: A `DQ` is not compared against itself.*

`filterWindowsForFiles(numberOfOverlaps, INJ_CAT)` Filters the windows listed in *self*.`overlappingWindows` for the purpose of writing to file. Applies the following filters:

1. Filters out category 1 windows, which are not written to file overlap file.
2. Separates category 3 windows, whose overlaps are to be written to a separate file.
3. Sorts the `overlappingWindows` to arrange alphabetically by `dqname` and in ascending `category` value.

`writeOverlappingWindows(ifo, numberOfOverlaps, INJ_CAT)` Writes `overlappingWindows` to xml, uses `filterWindowsForFiles`.

`filterOverlappingTriggers(isClustered, numberOfOverlaps, INJ_CAT)` Returns a list where each element in the list a is python `tuple` consisting of a trigger and the window that contains it, where the window also contains the information regarding which other windows overlap it.

`writeOverlappingTriggers(ifo, isClustered, numberOfOverlaps, INJ_CAT)` Writes the set of Overlapping (or Isolated) triggers to xml, uses `filterOverlappingTriggers`.

`createTriggerDictionary(isClustered, maxNumberOfOverlaps)` Creates a python `dictionary`, where each ***`key`*** is a `tuple` of the each trigger from `filterOverlappingTriggers` and number of `overlapping` windows currently being processed, and the ***`value`*** is the list of all the `overlapping` windows in which the (***`key`***) trigger is associated. *Note that a trigger is considered to be overlapping if it occurs within an overlapping window, therefore all other windows with which the initial window overlaps are associated with that trigger.*

C.4.2 class `PlotTriggers`

`PlotTriggers(object)` *The PlotTriggers class is used to generate plots of the isolated and overlapping triggers. Initialized by `PlotTriggers(dataSet, triggerSetDictionary)`.*

Each of the following is an attribute of `PlotTriggers`:

`triggerSetDictionary` Python `dictionary` *`keyed`* by trigger and number of `overlapping` windows where the *`values`* are lists of all the `overlapping` windows in which the (*`key`*) trigger is associated.

`dataSet` The **`DataSet`** from which this `PlotTriggers` instance is derived.

`unclusteredTriggers` The list of unclustered triggers that belong to the `DataSet` on which this `PlotTriggers` instance is derived.

The following are methods of the class `PlotTriggers`:

`getTriggerSetDictionary` Returns `triggerSetDictionary`

`getDataSet` Returns `DataSet`.

`getUnclusteredTriggers` Returns `unclusteredTriggers`.

`makeTriggerPlots` Verifies that all windows containing a particular trigger are included in the plot, then uses `createTriggerPlot` to create the plot of the trigger. *Note: Verifying window inclusion is necessary since a `DQ` is not compared against itself in determining `overlappingWindows`, but a `DQ` can generate vetoes of multiple categories with different `PaddedWindows`.*

`createTriggerPlot(trigger, containingWindows, originalWindows, additionalWindows)` Creates the plot of the isolated or overlapping trigger against the background of `unclusteredTriggers`, with all boundaries of all windows that contain that trigger.

getIsolatedTriggerWindowBoundaries(trigger, windows) Determines the boundary axes of an isolated or overlapping trigger plot. Used by `createTriggerPlot` in determining background triggers to include in plot.

C.5 `VetoedAndNonvetoedTriggers.py` and `outliers.py`

C.5.1 class `VetoedAndNonvetoedTriggers`

VetoedAndNonvetoedTriggers(object) *The VetoedAndNonvetoedTriggers class is used to identify the vetoed and non-vetoed triggers, sort by veto category and generate histograms relating the vetoed and non-vetoed triggers. Initialized by* *`VetoedAndNonvetoedTriggers(dataSet, isClustered, isPadded)`.*

Each of the following is an attribute of `VetoedAndNonvetoedTriggers`:

isPadded `True` (`False`) value indicating whether `PaddedWindows` (`UnpaddedWindows`) are to be used.

isClustered `True` (`False`) value indicating whether *clustered* (*unclustered*) triggers are to be used.

dataSet The **DataSet** from which this `VetoedAndNonvetoedTriggers` instance is derived.

triggers triggers from `dataSet.clusteredTriggers` or `dataSet.unclusteredTriggers` depending on value of `isClustered`

vetoedTriggers A python *`dictionary`* of vetoed triggers where the keys are the `categoryKeys`, and the values are lists of triggers vetoed by DQ windows of those veto categories.

nonVetoedTriggers The set of all triggers not in `vetoedTriggers` after each `categoryKey` is applied.

categoryKeys A list of tuples of categories and combinations of categories for DQ windows. *Note:* `categorykeys` *are a list, as follows: [(1,), (2,), (3,), (4,), (5,), (1,2), (1,2,3), (1,2,3,4), (1,2,3,4,5)].*

The following are methods of the class `VetoedAndNonvetoedTriggers`:

writeFiles Creates subdirectories, loops over `categoryKeys` callign `writeToFiles` to write triggers, calls `histogram`, creates `Outliers`, and calls `writeOutlyingTriggers` to write outlying triggers.

writeToFiles(subDirectory, categoryKey, vetoed) Writes `vetoedTriggers` (`nonVetoedTriggers`) to XML tables per category or categories by which the triggers are vetoed(survive vetoing).

histogram(subDirectory, categoryKey, binStep, triggerbins, triggerfreq) Generates histograms of `vetoedTriggers` and `nonVetoedTriggers` per category level.

C.5.2 class `Outliers`

Outliers(object) *The Outliers class is used to identify the outlying triggers. Initialized by* `Outliers(dataSet, properties, threshold, categoryKey, nonVetoedTriggers, isClustered, isPadded))`*.*

Each of the following is an attribute of `Outliers`:

dataSet The **DataSet** from which this `Outliers` instance is derived.

threshold The snr `threshold` on which this `Outliers` instance is based.

categoryKey The veto category or combination of veto categories from which this `Outliers` instance is derived.

nonVetoedTriggers The list of `nonVetoedTriggers`

isPadded `True` (`False`) value indicating whether `PaddedWindows` (`UnpaddedWindows`) are to be used.

isClustered `True` (`False`) value indicating whether *clustered* (*unclustered*) triggers are to be used.

outliers A python ***dictionary*** of outlying triggers where the ***keys*** are the `trigger.coarseTime` of the `trigger` objects, and the ***value*** of each is the `trigger` object which has the highest `trigger.snr` value of all `trigger` objects which correspond to that `trigger.coarseTime`.

nearbyDQWindows A python ***dictionary*** of nearby windows, keyed by the same `trigger.coarseTime` keys as `outliers`, with values that are a list of `DQ` windows whose boundaries are within the same `analyzedSegmentWindow` as the outlying trigger.

analyzedSegments A python ***dictionary*** of analyzed segments which is keyed by the same `trigger.coarseTime` keys as `outliers` and having values that are a list of analyzed segment windows (See **AnalyzedSegmentWindow** in appendix C.1.5) that contain that `trigger.coarseTime`.

scienceSegments A python ***dictionary*** of science segments, keyed by the same `trigger.coarseTime` keys as `outliers`, with values that are a list of science windows (See **getScience** in appendix C.1.1) that contain that `trigger.coarseTime`.

The following are methods of the class `Outliers`:

getWindowWithMaxEndTime(listOfWindows) Returns the **Window** object with latest end time from the list given as the argument.

getWindowWithMinStartTime(listOfWindows) Returns the **Window** object with earliest start time from the list given as the argument.

checkIfCloseBefore(triggerCoarseTime, maxBeforeWindow) Confirm that the (nearest) DQ window ending before triggerCoarseTime is within the same analyzedSegments window containing that triggerCoarseTime.

checkIfCloseAfter(triggerCoarseTime, minAfterWindow) Confirm that the (nearest) DQ window beginning after triggerCoarseTime is within the same analyzedSegments window containing that triggerCoarseTime.

getNearbyDQWindows Determines and returns nearbyDQWindows.

writeOutlyingTriggers Writes the outliers to an XML table.

plotOutlyingTriggers Generates plots of the outlying triggers in outliers.

APPENDIX D

APPENDIX D

FUNCTIONS FOR SETTING UP THE ENVIRONMENT

D.1 `initialize.py`

Initializes log and properties object.

D.2 `configuration.py`

D.2.1 `def parse_options`

`parse_options` *The parse_ options function parses the user options. Called by* ***`parse_options()`****.*

D.2.2 `def getConfiguration`

`getConfiguration` *The getConfiguration function, called by* ***`getConfiguration(log)`***:

Creates properties object with configuration file values.

Deletes existing directory if clean option selected.

Updates properties dictionary with command line options

Verifies user tasks and tasks associated with user assigned tasks.

Applies default properties values, optionally print properties to console.

Verifies Properties requirements.

Returns **`Property`** object used in this execution of `DQTunePipe`.

D.2.3 `def LoadConfigFile`

`LoadConfigFile` *The LoadConfigFile is used to load the configuration parameters specified by the user via a configuration file. Called by* ***`LoadConfigFile(file, properties=Properties()`****.*

Returns the **`Property`** object with configuration file values applied.

D.2.4 class `DefaultSectionHeader`

`DefaultSectionHeader(object)` *The DefaultSectionHeader class is used to create a generic section header for loading the configuration file. Initialized by* *`DefaultSectionHeader(fp)`*.

Each of the following is an attribute of `DefaultSectionHeader`:

- **`fp`** file pointer, (an open file for writing).
- **`sectionHead`** String, "[Configuration Options]" to serve as default section to use with `LoadConfigFile`.

Methods of the class `DefaultSectionHeader`:

- **`readline`** Returns the default section header as the a line read from the open file to use with `ConfigParser`.

D.3 `config_rules.py`

D.3.1 class `Rule`

`Rule(object)` *The Rule class is an abstract class for defining Rule objects.*

Methods of the class `Rule`:

- **`print_value`** Prints the value of the `properties` attribute for this **`Rule`**.

`apply` Abstract method which tests the rule.

`get_fail_message` Abstract method which returns the failure message for this rule.

`_has_value` Abstract method which determines if a value has been assigned.

D.3.2 class `Required`

`Required(Rule)` *The Required class extends **`Rule`** to require that the specified property contains a value. Initialized by `Required(propertyA)`.*

Attributes of `Required`:

`propertyA` The property attribute.

The following are methods of the class `Required`:

`apply` Apply Rule: Return True if `propertyA` has a value that is not `None`.

D.3.3 class `OrRequired`

`OrRequired(Rule)` *The OrRequired Rule is used to determine if at least one of two required properties attributes have values. Initialized by `OrRequired(propertyA, propertyB)`.*

Attributes of `OrRequired`:

`propertyA` The first property attribute.

`propertyB` The second property attribute.

Methods class `OrRequired`:

`apply` Apply Rule: Return `True` if `propertyA` or `propertyB` has a value that is not `None`.

D.3.4 class `IfThenRequired`

`IfThenRequired(Rule)` *The IfThenRequired Rule is used to determine that if one properties attribute is set then a second property attribute is also set. Initialized by `IfThenRequired(propertyA, propertyB)`.*

Attributes of `IfThenRequired`:

`propertyA` The first property attribute.

`propertyB` The second property attribute.

Methods of the class `IfThenRequired`:

`apply` Apply Rule: Return `True` if `propertyA` is not set or `propertyA` is set and `propertyB` is also set; otherwise returns `False`.

D.3.5 `def test_rules`

`test_rules` *The test_rules method is used to verify that user has provided all required input parametersand is called by `test_rules(rules, properties)`.*

Returns no output: Prints an error message to screen if a rule is broken, i.e. a required user input has not been provided.

D.4 `manageData.py`

D.4.1 `def has_dq_xml_files`

`has_dq_xml_files` *Called by `has_dq_xml_files(ifo)`.*

Return `True` if the DQ XML directory `properties.DQ_XML_DIRECTORY` has any appropriate files, else return `False`.

D.4.2 `def get_dq_xml_data`

get_dq_xml_data *Initialized by* *`get_dq_xml_data(ifo)`*.

Return `True` if DQ data must be retrieved, else return `False` is DQ data already present.

D.4.3 `def has_intermediate_trigger_data`

has_intermediate_trigger_data *Initialized by* *`has_intermediate_trigger_data(ifo, format)`*.

Return `True` if the summary data directory has appropriate intermediate files.

D.4.4 `def has_raw_trigger_data`

has_raw_trigger_data *Initialized by* *`has_raw_trigger_data(ifo, format)`*.

Return `True` if the trigger directory (`properties.TRIGGER_FILES[ifo]`) has appropriate files, else return `False`.

D.4.5 `def get_raw_trigger_data`

get_raw_trigger_data *Initialized by* *`get_raw_trigger_data(ifo, format)`*.

Retrieves the raw trigger data.

D.5 `rawData.py`

D.5.1 `def getDQxmlsALL`

getDQxmlsALL *The getDQxmlsALL method writes the DQ XML files (output returned by ligolw_segment_query) to the location identified by* *`properties.DQ_XML_DIRECTORY`*. *Called by* *`getDQxmlsALL(ifo)`*.

Returns no output.

D.5.2 `def getDQNamesFromligolwDQTools`

`getDQNamesFromligolwDQTools` *The getDQNamesFromligolwDQTools method writes the DQ output returned by ligolw_dq_query to a customized file, (`properties.DQ_LIST_FILE`) and keeps this data for use by all IFOs. Called by `getDQNamesFromligolwDQTools(ifo)`.*

Returns `outDQList`: A unique list of all DQ names in `properties.DQ_LIST_FILE` for a specific `ifo`.

D.5.3 `def copyTriggers`

`copyTriggers` *The copyTriggers method is used to copy the triggers from the location identified by `properties.TRIGGER_SOURCE_DIRECTORIES` to the locations specified for each interferometer by `properties.TRIGGER_FILES[ifo]`. Called by `copyTriggers(ifo, fileFormat)`.*

Returns no output.

D.5.4 `def copyVetodefinerFile`

`copyVetodefinerFile` *This method puts a copy of properties.VETODEFINERFILE_FULLPATH in the local directory when executing on padded data (`properties.PADDINGS` includes `True` value). Called by `copyVetodefinerFile()`.*

Returns no output.

D.6 `summaryDataFiles.py`

D.6.1 `def time_and_snr`

`time_and_snr` *The time_and_snr method creates a `triggerList` containing the end time and snr of each trigger in the XML table identified by the `dataTag`. Called by* *time_and_snr(ifo,reader,dataTag)*.

Returns triggerList: List of Trigger objects.

D.6.2 def segment

segment *The segment method creates segmentList containing the duration of each trigger in the xml table identified by groupTag. Called by segment(IFO,reader,groupTag).*

Returns segmentList: List containing Trigger durations.

D.6.3 def make_intermediate_trigger_data

make_intermediate_trigger_data *The make_ intermediate_ trigger_ data method creates the clustered and unclustered trigger text files and creates the analyzed segment text file from the unclustered XML files. Called by make_intermediate_trigger_data(ifo, clustered).*

Returns no output.

D.7 createDataSet.py

D.7.1 def createDataSet

createDataSet *createDataSet is a method that initializes the (DataSet) and is called by createDataSet(ifo)*:

- Instantiates DataSet object, dataSet, with ifo, properties.INITIAL_START, properties.INITIAL_END, properties.DQ_SCIENCE_FLAGS[ifo].
- Loops over DQ xml files in properties.DQ_XML_DIRECTORY; uses XMLTableReader to read each DQ xml file. Adds DQ object, dq created from name of DQ in xml file to dataSet.
- Adds Window object created from start and end times of DQ segments in DQ xmls to dq.

Reads `properties.VETO_DEFINER_FILE` via `XMLTableReader` to extract padding definitions and adds `Padding` objects for each `dq` to `dataSet`.

Adds list of clustered and unclustered `Trigger` objects to ***`dataSet`*** via `getTriggersFromFile`.

Returns `dataSet`: The instance of `DataSet` object. See Appendix C.1.

D.7.2 `def getTriggersFromFile`

`getTriggersFromFile` *The getTriggersFromFile method creates a list of* **`Trigger`** *object instances in the unclustered (clustered) trigger text file identified by* ***`properties.UNCLUST_TRIGGER_FILE (properties.CLUST_TRIGGER_FILE)]`***. *Called by* ***`getTriggersFromFile(file)`***.

Returns `triggers`: List of `triggers` for use by `createDataSet`.

D.8 Additional materials included with `DQTunePipe` that are not python modules.

D.8.1 `exampleConfigurationFile.txt`

An example configuration file. The user may use this as a template from which to develop a working configuration file for his environment.

D.8.2 `mod_ligolw.xsl`

Default stylesheet assigned to ***`properties.LIGO_LW_XSL`*** *for use in viewing various* ***`DQTunePipe`*** *output in browser.*

APPENDIX E

APPENDIX E

BACKGROUND AND SUPPLEMENTAL CLASSES AND FUNCTIONS

E.1 `XMLTableReader.py`

E.1.1 class `XMLTableReader`

`XMLTableReader(object)` *The XMLTableReader class is used to read xml data from a* `ligolw` *xml. Initialized by* `XMLTableReader(fileName)`*. A side note on import lines:* `from elementtree.ElementTree import parse` *must be used for Python version 2.4.3 or earlier (such as on ligo.caltech.edu), but should be replaced with* `from xml.etree.cElementTree import parse` *on Python versions 2.5 and later.*

Each of the following is an attribute of `XMLTableReader`:

`tables` List of tables elements in a `ligolw` XML, identified by `Table` and parsed via class `XMLTable`.

The following are methods of the class `XMLTableReader`:

`getTables` Returns **`tables`**.

`getTable` Takes `tableName` as an argument, returns the individual `table` identified by the given `tableName`.

E.1.2 class `XMLTable`

`XMLTable(object)` *The XMLTable class is used to identify the contents of a particular `Table` element in a `ligolw` XML. Initialized by `XMLTable(tableElement)`.*

Each of the following is an attribute of `XMLTable`:

`tableName` The name of the `Table` element

`columns` The columns of the `Column` elements for the table

`streams` The stream of the `Stream` elements for the table

The following are methods of the class `XMLTable`:

`getName` Returns **`tableName`**.

`getColumns` Returns **`columns`**.

`getRawStream` Returns **`streams`**, as one continuous string of text.

`getStream` Returns data in **`streams`**, as a list of python dictionary items, keyed by the name of the column, where the value is the associated content from the stream.

E.1.3 class `XMLTableColumn`

`XMLTableColumn(object)` *The XMLTableColumn class is used to identify contents of `Column` elements inside `Table` elements. Initialized by `XMLTableColumn(columnElement)`.*

Each of the following is an attribute of `XMLTableColumn`:

`name` The name of the `Column` element

`type` The type of the `Column` element

The following are methods of the class `XMLTableColumn`:

`getName` Returns the **`name`**.

getType Returns the **type**.

getValue Takes a `valueString` as argument, and returns the string, `valueString`, which is the content associated with column instance.

E.1.4 class `XMLTableStream`

XMLTableStream(object) *The XMLTableStream class is used to identify contents of xml streams in the tables of `ligolw` xmls. Initialized by `XMLTableStream(streamElement)`.*

Each of the following is an attribute of `XMLTableStream`:

name The name of the `Stream` element

type The type of the `Stream` element

delimiter The delimiter of `Stream` stream element

data The contents of the `Stream` element.

The following are methods of the class `XMLTableStream`:

getName Returns the **name**.

getType Returns the **type**.

getDelimiter Returns the **delimter**.

getData Returns the **data**.

E.2 `xmlColumnTag.py`

E.2.1 `def xmlColumnTag`

xmlColumnTag *The xmlColumnTag function loops over a list of tuples containing the name and type of items in a ligolw column and adds them to a supplied `dataTag`. Called by `xmlColumnTag(doc, dataSetTag, tupleList)`.*

Returns `dataTag`: Used for creating `ligolw` column elements that identify contents of XML tables.

E.3 `utility_box.py`

E.3.1 `def makeDirectory`

`makeDirectory` *The makeDirectory method verifies that a directory named **`newdir`** does not exist, and creates **`newdir`** and its necessary parent directories. Called by **`makeDirectory(newdir)`**.*

Returns no output.

E.3.2 `def median`

`median` *The median method sorts a list of numbers and determines the median value. Initialized by **`median(list)`**.*

Returns `medianValue`: The median value of input `list`.

E.3.3 `def parseBoolean`

`parseBoolean` *The parseBoolean turns a string "True" or "False" into a boolean* `True` *or* `False` *value. Initialized by **`parseBoolean(b)`**, where **`b`** is a string.*

Returns `True` for any string whose first character is `t`, `y`, or `1`.

Returns `False` for all other strings.

E.3.4 `def uniquifyList`

`uniquifyList` *The uniquifyList function returns a sorted list comprised of the unique elements in the input list. Called by **`uniquifyList(originalList)`**.*

Returns `uniqueList`: A sorted list of unique elements.

E.3.5 `def cluster_triggers`

`cluster_triggers` *The `cluster_triggers` method groups `Trigger` objects into a cluster so that all `Trigger` objects within the cluster are no more than `delta` apart in time. For each cluster it selects the `Trigger` object with the max snr to represent the cluster. A list of these representative triggers is returned. Called by `cluster_triggers(triggers, delta)`.*

Returns `cluster_triggers`: List of `Trigger` objects

VITA

Brooke Anne Rankins

Bachelor of Arts, Chemistry University of Mississippi, May 2001

Vice President of the Society of Physics Students, University of Mississippi Chapter Fall 2005

Sigma Pi Sigma University of Mississippi, May 2, 2006

Bachelor of Science, Physics University of Mississippi, May 2006

LIGO Science Collaboration Member , University of Mississippi, 2008-present

NASA Student Space Ambassador , Mississippi, January 2011-July 2011