

Searching for correlations in global environmental noise within the context of gravitational wave detection

A thesis written by

Karl Wette

being an account of research undertaken in the

**Centre for Gravitational Physics
The Department of Physics, The Faculty of Science
The Australian National University**

under the supervision of

Dr Susan Scott

submitted in partial fulfillment of the requirements for the degree of
Bachelor of Science with Honours in Theoretical Physics



Canberra, Australia

29th October, 2004

Acknowledgements

Right. It is now a few seconds past the thesis due date, and I have now gone without sleep for over 32 hours. As I type this I see random points of light piece my vision from out of nowhere. What to do? Nothing I suppose, except to make the customary acknowledgements:

To my supervisor, Susan Scott: thank you for presenting me with what has been at times an incredibly frustrating (as I'm sure you'd agree!) but in the end a most interesting and challenging honours project. Thank you for your continued support and encouragement throughout. I also gratefully acknowledge the financial assistance of The Australian National University through an ANU Honours Scholarship.

I am very thankful for the efforts of Antony Searle and Tom Kobialka, in joining with me in pitched battles against the demon that is Computer Software. In particular, thanks to Tom for his help with FrameL and LDR, 'Surly' for his programming help and Linux networking bag-o-tricks, and bronze stars to you both for warding off the evil spammers that attempted to take over the cluster.

To Albert Lazzarini, Benoit Mours, and other scientists involved with PEM data acquisition at LIGO and VIRGO: thank you for helping an insignificant honours student complete his project.

Angela, Ben, Lydia, Andrew, and Antony: thanks for the many welcomed distractions and discussions throughout the year, and for your help in proofreading my thesis (even when it resulted in more work). To Amy, Ben, Phil, and Anthony at "Burgie", cheers for helping me make the rest of my life more enjoyable. I also acknowledge all the random people I have met and corresponded with in Australia over the past year; without knowing it, you helped me keep a grasp on my own sanity just that little bit longer.

To my wonderful family, for your continued love and support, weekly updates from New Zealand, and wise counsel during difficult times: I would never have survived without you.

And finally, to all the myriad problems which have dogged my progress this year: broken instruments, software hacks that work one minute and not the next, bugs that can never be found, deficient operating systems, obtuse programs, stupid programming languages, tropical cyclones, and everything else, I can only say this:

I am still here.

That is all.

Declaration

This thesis is an account of research undertaken between February and October 2004, in The Centre for Gravitational Physics, The Department of Physics, The Faculty of Science, The Australian National University, Canberra, Australia. Except where appropriately acknowledged, the material presented in this thesis constitutes my own original work, and furthermore has not been previously submitted in whole or in part for a degree at any university.

Karl Wette
Canberra, Australia
29th October, 2004

Abstract

The high sensitivity required of interferometric gravitational wave detectors is such that environmental noise becomes a very important consideration. Local environmental noise may be partially or wholly eliminated by cross-correlating data from two widely displaced detectors; such a strategy, however, will not be immune to environmental noise that is global in nature. Thus the characterisation of global environmental noise is of vital importance to the detection of gravitational waves.

In this project the author will attempt to gain some understanding of physical environment noise, in particular whether it can be global as well as local in nature, by investigating the correlation of widely displaced physical environment monitoring stations. We found that, though there were no immediately significant correlations between the detectors, the unknown physical origin of the observed correlations would certainly be worthy of further research.

Contents

Acknowledgements	iii
Declaration	v
Abstract	vii
Prologue	1
1 Introduction	3
1.1 Large scale interferometric gravitational wave detectors	3
1.2 Review of physical environment noise investigations	4
1.3 Overview of this project	6
2 Data acquisition	7
2.1 A physical environment monitoring station	7
2.1.1 The original setup	7
2.1.2 Additional hardware	9
2.2 The data acquisition software	9
2.2.1 The original setup	9
2.2.2 Porting libraries	10
2.2.3 Code interfaces	10
2.2.4 Virtual circuitry	15
2.2.5 Uploading files	18
2.3 The data storage system	19
2.3.1 The original format	19
2.3.2 Reprocessing	19
2.4 Data acquisition in practice	20
2.4.1 Locally	20
2.4.2 From LIGO	20
2.4.3 From VIRGO	21
3 Data analyses	23
3.1 Long timescale analyses	23
3.1.1 The autocorrelation	23
3.1.2 The correlation	24
3.1.3 The power spectrum	24
3.1.4 The cross spectrum and the coherence	27
3.2 Short timescale analyses	28
3.2.1 What is a transient event?	28
3.2.2 The power spectrogram and the cross spectrogram	29
3.2.3 The search algorithm	29
3.2.4 The difference-in-mean test	30

3.2.5	The Student paired t-test	30
3.2.6	The Wilcoxon signed rank test	31
3.2.7	The Kolmogorov-Smirnov test	31
3.2.8	The Wilcoxon-Mann-Whitney rank sum test	31
4	Data processing	33
4.1	Some technologies	33
4.1.1	Parallel programs	33
4.1.2	Message-passing	33
4.1.3	MPI	34
4.1.4	LAM/MPI	35
4.1.5	FFTW	35
4.1.6	C	35
4.1.7	The GNU Compiler Collection	35
4.2	A parallel data processing program	36
4.2.1	The header	36
4.2.2	The main subroutine	36
4.2.3	The collector subroutine	39
4.2.4	The processor subroutine	42
4.2.5	Implementation of the long timescale analyses	43
4.2.6	Implementation of the short timescale analyses	44
5	Results	45
5.1	Long timescale analyses	45
5.1.1	Power spectra	45
5.1.2	Autocorrelations	46
5.1.3	Correlations	57
5.1.4	Coherence	59
5.2	Short timescale analyses	65
6	Conclusion	71
A	Additional code	73
A.1	The parallel data processing program	73
A.1.1	The header	73
A.1.2	The main subroutine	75
A.1.3	The collector subroutine	77
A.1.4	The processor subroutine	79
A.1.5	Implementation of the long timescale analyses	81
A.1.6	Implementation of the short timescale analyses	84
A.1.7	Miscellaneous subroutines and functions	87
A.2	The MATLAB post-processing code	96
	Bibliography	105

Figures

2.1	The physical environment monitoring station	8
2.2	The LabVIEW data acquisition software diagram	16
2.3	The LabVIEW data acquisition software front panel	17
2.4	Timeline of the physical environment monitoring station data; the shaded areas indicate the times for which data exists in the Mass Data Storage System.	19
4.1	Example for four physical environment monitoring stations of the structure of the collector-to-processor/cross-processor communicators. Numerals in brackets refer to ranks in the new communicators; those without refer to ranks in MPI_COMM_WORLD.	38
5.1	Summary of the performed data processing.	45
5.2	Power spectra of the ANU horizontal and vertical seismometers.	47
5.3	Power spectra of the ANU mains voltage monitor and magnetic field sensors.	48
5.4	Power spectra of the LHO horizontal and vertical seismometers.	49
5.5	Power spectra of the LHO mains voltage monitor and magnetic field sensors.	50
5.6	Power spectra of the LLO horizontal and vertical seismometers.	51
5.7	Power spectra of the LLO mains voltage monitor and magnetic field sensors.	52
5.8	Power spectra of the VIRGO horizontal and vertical seismometers.	53
5.9	Power spectra of the VIRGO mains voltage monitor and magnetic field sensors.	54
5.10	Autocorrelation of the ANU horizontal and vertical seismometers.	55
5.11	Autocorrelation of the LHO horizontal seismometer.	56
5.12	Autocorrelation of the LLO vertical seismometer.	56
5.13	Autocorrelation of the LLO mains voltage monitor.	56
5.14	Autocorrelation of the VIRGO horizontal and vertical seismometers.	57
5.15	Correlation of ANU-LHO mains voltage monitors.	58
5.16	Correlation of the ANU-VIRGO horizontal and ANU-LLO vertical seismometers.	58
5.17	Correlation of the LHO-LLO horizontal and vertical seismometers.	60
5.18	Correlation of the LHO-LLO magnetic field sensors.	61
5.19	Coherence of the ANU-VIRGO mains voltage monitors.	62
5.20	Coherence of the ANU-LHO, ANU-LLO and LHO-LLO magnetic field sensors.	63
5.21	Coherence of the LHO-LLO, LHO-VIRGO and LLO-VIRGO magnetic field sensors.	64
5.22	Bursts in seismometer spectrograms.	66
5.23	More bursts in seismometer spectrograms.	67
5.24	Bursts, gaps, and wandering lines in magnetic field sensor spectrograms.	68
5.25	Spectrographic “phase changes” in seismometer and magnetic field sensor spectrograms.	69

Listings

2.1	The OpenFile code interface node subroutine	11
2.2	The OpenFrame code interface node subroutine	12
2.3	The OpenFrame code interface node subroutine (continued)	12
2.4	The OpenFrame code interface node subroutine (continued)	12
2.5	The AddData code interface node subroutine	13
2.6	The AddData code interface node subroutine (continued)	13
2.7	The AddData code interface node subroutine (continued)	14
2.8	The CloseFrame code interface node subroutine	14
2.9	The CloseFile code interface node subroutine	14
2.10	The MoveFiles AppleScript	18
4.1	The main subroutine	36
4.2	The main subroutine (continued)	36
4.3	The main subroutine (continued)	36
4.4	The main subroutine (continued)	37
4.5	The main subroutine (continued)	38
4.6	The main subroutine (continued)	39
4.7	The collector subroutine	39
4.8	The collector subroutine (continued)	39
4.9	The collector subroutine (continued)	39
4.10	The collector subroutine (continued)	40
4.11	The collector subroutine (continued)	40
4.12	The collector subroutine (continued)	41
4.13	The collector subroutine (continued)	41
4.14	The collector subroutine (continued)	41
4.15	The processor subroutine	42
4.16	The processor subroutine (continued)	42
4.17	The processor subroutine (continued)	42
4.18	The processor subroutine (continued)	43
A.1	The header	73
A.2	The main subroutine variable declarations	75
A.3	The main subroutine construction of the collector-to-processor/cross-processor communicators	75
A.4	The main subroutine processing of the channels file	75
A.5	The main subroutine calls to the collector and processor subroutines	76
A.6	The main subroutine cleanup	76
A.7	The collector subroutine variable declarations	77
A.8	The collector subroutine initialisation	77
A.9	The collector subroutine synchronisation of starting times	78
A.10	The collector subroutine transmission of data	78
A.11	The collector subroutine cleanup	79
A.12	The processor subroutine variable declarations	79

A.13 The processor subroutine initialisation	79
A.14 The processor subroutine reception of data	80
A.15 The processor subroutine execution of the analyses	80
A.16 The processor subroutine cleanup	81
A.17 The doLongAuto subroutine	81
A.18 The doLongCorr subroutine	82
A.19 The doLongPower subroutine	83
A.20 The doLongCross subroutine	83
A.21 The doTrnsPower subroutine	84
A.22 The doTrnsCross subroutine	84
A.23 The doTrnsSearch subroutine	85
A.24 The algrCross subroutine	87
A.25 The algrCrossSqrd subroutine	87
A.26 The algrRead function	88
A.27 The algrWrite subroutine	88
A.28 The bufferAdd subroutine	89
A.29 The bufferFree subroutine	90
A.30 The bufferInit subroutine	90
A.31 The bufferShift subroutine	90
A.32 The error subroutine	90
A.33 The longFinal subroutine	91
A.34 The longInit subroutine	91
A.35 The qsort_double_abs function	92
A.36 The qsort_fftw_real function	92
A.37 The spctgmWrite subroutine	92
A.38 The sprintfalloc function	93
A.39 The statCompute subroutine	93
A.40 The strcatfalloc subroutine	94
A.41 The trnsFinal subroutine	95
A.42 The trnsInit subroutine	95
A.43 The MATLAB post-processing code	96

Prologue

Da könnt' mir halt der liebe Gott leid tun, die Theorie stimmt doch.
Then I would have been sorry for the dear Lord – the theory *is* correct.

— Einstein [1]

The general theory of relativity is one of the foremost accomplishments of human thought. It is a world where space and time, freed from the chains of the absolute, may merge, entwine, twist and contort themselves. It is the world of black holes, pulsating quasars, naked singularities, and the Big Bang. It is a world of a mathematical beauty appreciable only by theoretical physicists, yet it has inspired generations of non-physicists in art, literature, cinema, and beyond. Therefore it is small wonder that Albert Einstein, its chief architect, was so completely convinced of its truth – for surely, if such a world extended no further than our imagination, then indeed our reality would be the poorer of the two.

Despite Einstein's conviction, ever since the publication of the theory in 1915 physicists have endeavoured to verify its predictions by experiment. While the first such experiment, the observation by Eddington of the deflection of light by the Sun, and of the correct magnitude predicted by general relativity, was not only a success but caused widespread public sensation, subsequent attempts to repeat this observation were significantly less successful, and the accuracy of the first attempt was called into question. By 1960, it could be argued that general relativity was supported experimentally by only two verified predictions: the precession of the perihelion of the orbit of Mercury, to an accuracy of approximately 1%; and the deflection of light by the Sun, to an accuracy of then barely 50%. Attempts to measure the gravitational red-shift of light, from the Sun and from white dwarf stars, were inconclusive, and subsequently it was shown that it was not in fact a consequence of the complete theory [2].

The twenty years following 1960, however, brought about a golden age of experimental general relativity. Experiments attempted in the past were reborn with the help of new technologies and experimental techniques. A series of experiments by Pound, Rebka and Snider measured gravitational red-shifts in the laboratory to high precision for the first time, while others successfully measured the effect in solar spectral lines. The change in the rates of atomic clocks lifted aloft on jet aircraft and on satellites were successfully measured. Experiments performed by Eötvös and others tested the principle of equivalence, while others successfully bounced lasers off the Moon to measure the Nordtvedt effect, and radio waves off Venus to measure the time delay of light. A series of discoveries with great importance to general relativity were made in astronomy, such as the discoveries of the cosmic microwave background, of pulsars, of binary pulsars and their characteristic orbital period decrease, and of the first black hole candidate Cygnus-X1. At the same time a more rigorous framework was put in place for the experimental verification of general relativity and of alternate theories of gravity. In all experimental tests conducted so far, general relativity has without fail predicted the results within experimental error [2].

Gravitational waves are the subtle oscillations induced in the structure of space-time by

accelerating bodies, in an analogous manner to the production of electromagnetic radiation by accelerating charges. They have come to the fore in the last two decades, not only as a means of testing general relativity in regions of strong gravity, such as for example would exist around black holes, but as a whole new spectrum of radiation from the Universe whose properties are yet to be investigated. The introduction to astronomy over the past 60 years of observation in the radio, infrared, X-ray and gamma ray components of the electromagnetic spectrum have heralded untold and previously unthought-of discoveries, and it is hoped that gravitational wave astronomy, when realised, will have an equal if not greater impact.

The first attempt to directly detect gravitational waves was made by Weber around 1960, using the resonance of massive aluminium cylinders, but this attempt was ultimately unsuccessful [3]. Though the construction of the so-called resonant bar detectors has continued, in the 1970s a new technique, using the perturbation in the difference in arm lengths of laser interferometers, was first considered and a number of prototypes constructed [4, 3]. Laser interferometric detectors, when successfully implemented on a large scale, will be the most likely instruments to first succeed in detecting and measuring gravitational waves. The obstacles standing in the way of this first detection, however, remain substantial.

Introduction

Gravitational waves are small — very, very small. The average signal is not expected to perturb the difference in arm lengths of a large scale interferometric detector by much more than 10^{-18} metres. The sensitivity required to measure such a change in length is analogous to measuring the surface of the Earth to the precision of a square millimetre. It is therefore almost certain that any detectable gravitational wave signal will be deeply embedded in a mass of random noise, the principal cause of which will be the surrounding physical environment.

For the detection of a gravitational wave signal to be credible, it must be clearly demonstrated that the signal could not have arisen by chance from random noise. This point cannot be stressed enough. The reputation of the entire gravitational wave detection community depends on their confidence in the initial detection of gravitational waves, and their ability to instil that confidence in the wider scientific community. This cannot be achieved unless every other possible cause of the candidate gravitational wave signal has been convincingly eliminated. The most likely source of potential candidate gravitational wave signals is physical environment noise; it is therefore essential to the future successful detection of gravitational waves to understand all of the physical environment noise sources that might hinder their detection.

1.1 Large scale interferometric gravitational wave detectors

In recent years, serious projects have been mounted to bring to fruition the detection of gravitational waves, by the construction of several large scale interferometric detectors with arm lengths up to several kilometres. To date they are the following:

- The Laser Interferometer Gravitational Wave Observatory (LIGO) [5] has constructed 3 detectors at 2 locations in the United States: a combined 4 kilometre and 2 kilometre arm length detector on the Hanford Nuclear Reservation near Richland, Washington State; and a 4 kilometre arm length detector near Livingston, Louisiana. The project is run by the California Institute of Technology and the Massachusetts Institute of Technology, in collaboration with numerous other universities around the world (including The Australian National University) through the LIGO Science Collaboration.
- The VIRGO Project [6] has constructed a 3 kilometre arm length detector near Pisa, Italy. It is a collaboration between the French National Centre for Scientific Research (CNRS¹) and the Italian National Institute for Nuclear Physics (INFN²).

¹*Centre national de la recherche scientifique*

²*Istituto Nazionale di Fisica Nucleare*

- The GEO600 project [7], a collaboration between the University of Hanover, the University of Glasgow, Cardiff University, and the Max Planck Institutes for Quantum Physics and for Gravitational Physics (the Albert Einstein Institute), has constructed a 600 metre arm length detector near Hanover, Germany.
- The TAMA300 project [8] has constructed a 300 metre detector in Tokyo, Japan. It is run by the Japanese National Astronomical Observatory in collaboration with several other Japanese research institutions.

To date no detector has acquired the sensitivity believed to be required for gravitational wave detection. The detectors most likely to eventually achieve such sensitivity have yet to become fully operational.

1.2 Review of physical environment noise investigations

Notwithstanding the importance of physical environment noise to gravitational wave detection, it is not yet a primary focus for research at the major observatories. Their chief concern is, as it should be, to eliminate or reduce the instrumental noise generated by the detector components as much as possible, in order to optimise the detection of gravitational waves. Within this context, however, some research has been undertaken into physical environment noise. The following is a review of such research conducted at the LIGO observatories.

Physical environment noise can be divided into two broad categories. Local physical environment noise is said to be localised within the surroundings of a particular gravitational wave detector only; it would therefore be uncorrelated between two widely displaced detectors. The LIGO project has built two widely displaced detectors precisely for the purpose of trying to identify local physical environment noise by cross-correlating the detectors. Such a strategy, however, will not be immune to potential global physical environment noise sources, which would be correlated between the detectors and thereby unidentifiable by this method.

Local physical environment noise sources can be further divided into three categories. One category is noise originating from the natural environment, due to either geophysical or atmospheric activity. Such noise cannot be controlled; the best that can be done is to monitor its effect on the gravitational wave detectors. If it is a continuous effect, one can then try to compensate for it in real time, as for example in the case of the sophisticated isolation stacks used to isolate the interferometer mirrors from seismic vibrations. If it is a transitory effect, one can try to develop methods for identifying and thus removing it after the fact. In the first case, extensive studies have been conducted to characterise seismic noise spectra and its effect on the detector instruments [9, 10]. In the second case several studies have identified the effects of various transitory phenomena on the detectors. For example, several earthquakes over the past four years have been clearly registered; other more local effects such as wind gusts have, on occasions, prevented the interferometer from achieving lock [11, 12]. Signals which appear to have been caused by lightning strikes have been registered by magnetic field sensors at both Hanford and Livingston detectors, but there was no correlation between them, and these signals have not been observed on other monitoring channels [13, 14, 15].

A second category is noise originating from machinery being operated in the environs of the detectors. Several studies have been made of these effects. For example, nearby construction work has at times prevented the Hanford interferometer from achieving lock

[12]. Tank fire on a military firing range near to the Hanford observatory was detected in seismic and acoustic monitoring channels [16]. Road traffic noise in the area around the Hanford observatory was subject to an extensive study, which determined the frequency of the resultant seismic noise to be a function of velocity and the spacing of the vehicles' axles [17, 12]. Heavy freight trains running on tracks near to the Livingston observatory are clearly identifiable in the seismic noise spectrum [18]. Giant cooling fans at a nuclear reactor near to the Hanford observatory were found to be the cause of 20% of the noise in some of the optical components [19]. A substantial study was made of a proposed wind farm project near to the Hanford observatory and its potential effect on the seismic background noise [20].

A third category is noise sources originating within the detector itself, or within its immediate environs. Unlike the previous two categories, this category of noise sources can in most cases be eliminated; hence it has attracted the most thorough investigation. The sources found to cause detectable effects on the detector gravitational wave channel range from cooling fans [19], air compressors [16], loose nuts and bolts [21], broken wires [19], local vehicles [16], and even the movement of people [16, 18, 15]! In addition to identifying and attempting to remove these noise sources, experiments have been made to deliberately generate acoustic, magnetic, radio and seismic noise in the vicinity of the detector instruments in order to characterise their response. This process identified several different couplings between many of the instruments [16, 22, 23].

The existence of truly global physical environment noise is a question whose answer is vitally important to the detection of gravitational waves. While local physical environment noise can be eliminated by cross-correlation between widely displaced detectors, global physical environment noise cannot. Some investigations have been made into correlations between the Hanford and Livingston detectors. Correlated transient bursts have been detected in the signals of seismometers, magnetic field sensors and mains voltage monitors, but further analysis suggested that they may have only been chance coincidences [13, 24, 14, 15]. Many continuous correlations were found to disappear when observed over long timescales [25]; many other correlations that persisted have since been identified and reduced or eliminated [26]. A few have still persisted, such as for example the corrections to the frequencies of the U.S. power grids [27, 28]. Many of the detected correlations may simply have been due to the similarity of some of the equipment at both sites, thus not implying any causal connection [21]. Some investigations have also been made into correlations between physical environment monitors at LIGO and at The Australian National University [29].

The investigation of long and short timescale correlations between instruments, such as were investigated above, is not usually found outside of the context of gravitational wave detection. Most experiments in physics do not require such techniques, and areas such as archaeology and climatology are generally more interested in simpler linear trends. Integration of a signal over a long timescale is often required in astronomy, but any correlation, such as between elements of an optical or radio telescope array, is known in advance by the geometry of the elements. And astronomers have the benefit of being able to accurately characterise their noise response instruments in the absence of the signal they are trying to detect, a convenience also unavailable to gravitational wave detectors. The only other project known to the author which requires long timescale searches for momentary correlations is the Search for Extra-Terrestrial Intelligence project [30].

In the future, gravitational wave detection will become gravitational wave astronomy, harnessing several large-scale interferometric detectors in a coherent analysis to detect the

precise location of gravitational wave sources. Such an observatory would, however, be severely restricted, perhaps even rendered impossible, by the presence of truly global physical environment noise. Thus, the search for the existence of global physical environment noise is very important, not only for the immediate needs of gravitational wave detection, but for the potential future of gravitational wave astronomy.

1.3 Overview of this project

In this project the author attempted to gain some understanding of physical environment noise, in particular the important question of whether it is global as well as local in nature, by investigating the correlation of widely displaced physical environment monitoring stations. The importance of the question of global physical environment noise is such that this project can only form an initial first step towards its answer; but it is hoped that it will provide sufficient insight in order to form a useful base for future research.

Chapter 2 details the physical environment monitoring station at The Australian National University, and some substantial modifications made to its original setup by the author. Chapter 3 gives an overview of the mathematical analyses used to search for long and short timescale correlations in the physical environment monitoring data from stations around the world. Chapter 4 presents the software written by the author to perform the analyses, using parallel programming techniques. Results from the completed analyses are presented in Chapter 5, and Chapter 6 will summarise the conclusions of the project and suggest future directions for this research.

Data acquisition

Monitoring apparatus sense disturbances in their surrounding environment. By their physical construction they interact with these disturbances, producing as output a continuous electrical signal in some way proportional to the disturbance. A data acquisition system must be able to acquire this output, perform any necessary calibration, and write the final digitised signal to permanent storage; and it must be able to perform this process repeatedly and reliably.

2.1 A physical environment monitoring station

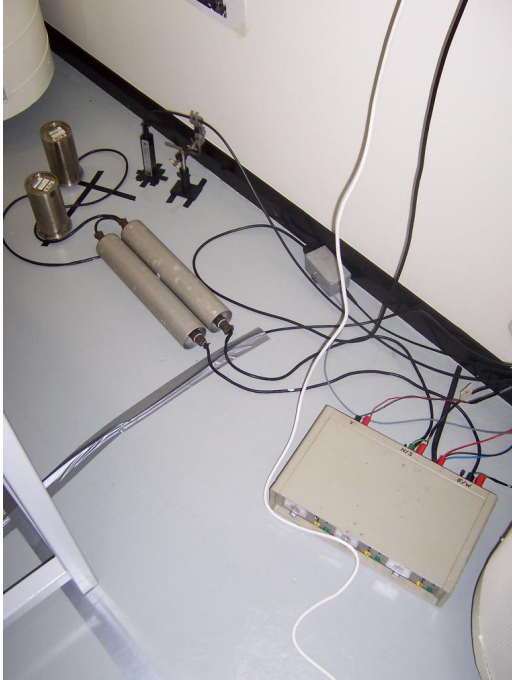
A physical environment monitoring station, originally set up by Benedict Cusack in 2002, is located in the Gravitational Wave Research Facility attached to the Department of Physics at The Australian National University (figure 2.1). This section details its original setup and some subsequent modifications made by the author.

2.1.1 The original setup

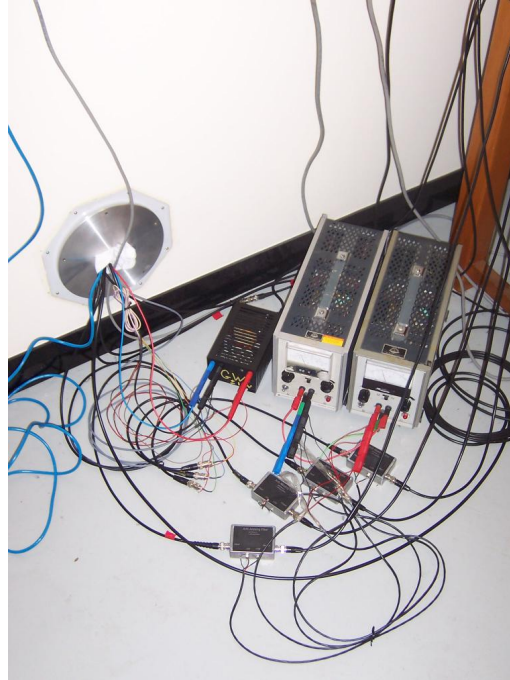
The station comprises 3 single-axis seismometers to measure seismic vibrations in the longitudinal (X), latitudinal (Y) and vertical (Z) directions; a Stefan Mayer Instruments [31] FL3-100 3-axis fluxgate magnetic field sensor to measure magnetic disturbances in the same directions; and a sensor custom-built in the Department of Physics to measure fluctuations in the mains voltage supply (figure 2.1(a)). The instruments are supplied by a ± 15 volt DC power supply (figure 2.1(b)). The seismometers are sampled at 256 Hz; the magnetic field sensor and mains voltage monitor are sampled at 2048 Hz. Acquisition is synchronised to the Global Positioning System (GPS) time standard used by all gravitational wave observatories.

Signal outputs are fed via Bayonet Neill-Concelman (BNC) coaxial cables to BNC jacks, the output wires from which are screwed into a National Instruments [32] CB-68LP terminal block. Also connected to the block are a TrueTime [33] XL-DC GPS time and frequency receiver, which is used to encode the GPS time at acquisition, and a Stanford Research Systems [34] DS345 30 MHz synthetic function generator, which is used to locally synchronise the GPS time receiver and the data acquisition card (figure 2.1(c)). Finally, the inputs to the terminal block are sent to a National Instruments PC-MIO-16XE-50 data acquisition card, installed on an Apple [35] PowerMac Blue-and-White G3 (figure 2.1(d)).

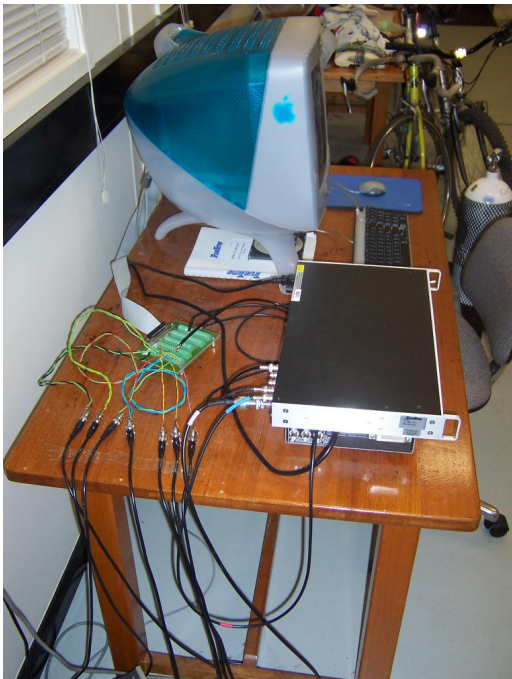
The principal problem with the monitoring station hardware concerned a single instrument: the vertical (Z) single-axis seismometer. This instrument was non-functional when the author first became involved, and a subsequent review of previously acquired data



(a) Inside the laboratory: the longitudinal (X) and latitudinal (Y) seismometers, the magnetic field sensor, the mains voltage monitor, the seismometer control box and tubular electronics units.



(b) Outside the laboratory: access hole to the laboratory, the monitors' power supply, the anti-aliasing filters' power supplies, and the anti-aliasing filters.



(c) The Apple Macintosh, the GPS time and frequency atop the synthetic function generator, the terminal block connecting the monitors to the data acquisition card.



(d) The GPS time and frequency atop the synthetic function generator, the Apple Macintosh running the LabVIEW data acquisition software.

Figure 2.1: The physical environment monitoring station
All identifications are specified clockwise from top left-hand corner of upright image.

found that it had been dysfunctional for most of the history of the station. Unfortunately it could not be repaired within the time frame of this project.

2.1.2 Additional hardware

During this project anti-aliasing filters were connected in series with the magnetic field sensor and mains voltage monitor signal outputs (figure 2.1(b)). The filters suppress all signals oscillations below ~ 1.6 Hz and above 900 Hz by -30 dB; see page 27 for a discussion of anti-aliasing. In order to connect them the magnetic field sensor was removed and BNC jacks were soldered to its output leads. BNC cables connected these to the inputs of the filters, and the outputs to a further set of BNC jacks, the output wires from which were screwed back into the terminal block. Power leads were soldered to the anti-alias filters and connected to a separate ± 15 volt power supply, consisting of 2 variable DC power supplies both set to +15 volts and connected in anti-series. It was found preferable to connect the filters to a separate power supply after finding that the filters, when connected to the same power supply as the monitoring instruments, not only overloaded the power supply but also induced severe distortion of the mains voltage monitor signal.

2.2 The data acquisition software

Data acquisition and storage is controlled using National Instruments' LabVIEW. LabVIEW is a sophisticated virtual instrumentation system that allows virtual electronic circuits to be built and executed in software. Each virtual component is represented visually by an icon; icons are connected together with software "wires" to form a circuit. LabVIEW includes common features of standard programming languages, such as conditional and iterative execution, file input/output and a variety of integer and floating point number, string, and array types. In addition a graphical user interface, using controls that mimic those of real instruments, can be added to allow data to be displayed and manipulated.

This section details the original data acquisition software and some substantial modifications by the author.

2.2.1 The original setup

The original data acquisition software was written by Benedict Cusack. It acquired data at 2048 Hz, down-sampled the seismometer data to 256 Hz, interpreted the GPS time signal, displayed the data graphically, and wrote the data using native file input/output functions to a set of intermediate files. An FTP server, NetPresenz, made the files available to a Perl script, written by Jon Smillie of the ANU Supercomputer Facility (ANUSF) [36], which periodically uploaded the files to the ANUSF's Mass Data Storage System (MDSS) [37], a robot-driven 1,200 terabyte tape library. A further script written by Benedict Cusack converted the intermediate files into Frame files using the FrameL library.

The Frame data format [38] is a standard agreed upon by LIGO, VIRGO and other projects for the exchange of gravitational wave related data. It is based upon a series of nested structures. At the top is the FrameH header structure, which contains basic information such as the origin, GPS start time and length of all data stored within that unit, or frame. It can also reference many other structures. The FrHistory structure contains information on any modifications such as data conditioning; the FrDetector structure contains basic information about the originating detector. There are many structures used to

store various types of processed and simulated data, as well as meta-data on any transient “events” detected by monitoring software. Actual experimental data is stored under the `FrRawData` structure, which in turn can reference several `FrAdcData` structures containing the raw analogue-to-digital signal from different channels. A Frame file itself may contain many `FrameH` structures. Additional information is written to the file in order to reconstruct the structures and data correctly irrespective of computer architecture or operating system.

The `FrameL` library [39] implements the Frame format in C [40]. It provides functions to read in and write out Frame files in various ways, to iterate through the frames in a file, to compress, tag and merge frames, and to create and extract various structures and their associated data.

2.2.2 Porting libraries

During this project significant modifications were made to the acquisition software. It was thought desirable to eliminate the intermediate files and additional conversion scripts, and instead to write the data directly as Frame files, thus making it more readily available for immediate use.

The first step was to port the `FrameL` library to the Macintosh operating system `MacOS 9.2.2`. Since `FrameL` is available as source code this involved only recompilation. The compiler used was part of the Apple Macintosh Programmers’ Workshop (MPW), a free suite of compilers, tools, scripts and reference material for writing and developing programs on the Macintosh. Its `MrC` compiler was used to recompile the `FrameL` source code. This was generally straightforward since the `FrameL` library only relies upon standard ANSI C libraries and includes additional required libraries as source code. A couple of problems were encountered, however. In dealing with these problems it was decided not to modify the `FrameL` source code itself, since such modifications would then have to be re-implemented should the `FrameL` library be upgraded.

A minor problem was that the `FrameL` library by default implements file input/output using UNIX system calls, which are unavailable under `MacOS 9.2.2`. It can also implement the same functionality, however, using standard C library functions; this was invoked by defining the `FRIOCFILE` preprocessor symbol, and creating blank files in place of some of the UNIX system header files referenced by the `FrameL` library.

A more major problem was encountered later. The `FrameL` library follows the UNIX convention of assuming that the names of files and directories will not contain any spaces. This is however not the case under `MacOS`; for example, “`Macintosh_HD`” is the default name for the Macintosh startup disk. Thus any attempt to specify the name and path of Frame files on the startup disk using the `MacOS` path convention, for example “`Macintosh_HD:LabVIEW_5.1:K-776999700-900.gwf`”, resulted instead in all files being written on a single file called “`Macintosh`” in the default directory. This problem was eventually circumvented simply by specifying only the file names, which do not contain spaces, and not their desired paths, which may have contained spaces. This resulted instead in all files being written to an undesired directory; the solution to this latter problem is discussed on page 18.

2.2.3 Code interfaces

The next step was to interface between the `FrameL` library and `LabVIEW`. This was accomplished using `LabVIEW` code interface nodes. A `LabVIEW` code interface node is a

LabVIEW component which directly executes an external subroutine written in C using a special interface. The subroutine can be passed arguments from LabVIEW and can modify and return those arguments; it can also call external C functions and reference C libraries such as FrameL. The subroutines are compiled and created using special tools that LabVIEW provides for a number of standard compilers, including MPW.

Five code interface nodes were written. Below are summaries of their respective function and implementation, along with images of their representative LabVIEW icons¹, which show the input arguments to the code interface node on the left-hand side of the icon, and the return arguments on the right-hand side.

The OpenFile code interface node opens a Frame file. Its input arguments specify the desired data compression algorithm (choices include combinations of a differential, zero suppression and gzip algorithms), the compression strength of the gzip algorithm (if invoked by the first argument), and the length of the file, expressed as the total number of seconds to be stored. It returns a raw pointer to the internal FrameL representation of the open file. The underlying subroutine is:

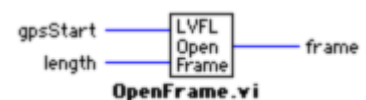


Listing 2.1: The OpenFile code interface node subroutine

```
#include "extcode.h"
#include "hosttype.h"
#include "FrameL.h"
#include "FrVect.h"
#define frameErr 999
CIN MgErr CINRun(int32 *compress, ulnt16 *gzipLevel, int32 *length, ulnt32 *file) {
    *file = (ulnt32) FrFileONewM("K", (int) *compress, NULL, (int) *length);
    if (*file == NULL) {
        DbgPrintf("FrFileONewM failed: %s\n", FrErrorGetHistory());
        return frameErr;
    }
    FrFileOSetGzipLevel((FrFile*) *file, (unsigned short) *gzipLevel);
    return noErr;
}
```

The first 4 lines direct the compiler to include header files required by LabVIEW and FrameL. The 6th line defines the special interface required by LabVIEW. The remaining lines open the file using the convenient FrameL function FrFileONewM, which automatically generates appropriate Frame file names and will transparently open a new file once the current file becomes full.

The OpenFrame code interface node creates a frame within which data can be stored. Its arguments specify the GPS time to associate with the frame and the length in seconds of the data to be stored. The GPS time is specified as the number of seconds since the Coordinated Universal Time (UTC)² date January 6th



¹The acronym “LVFL” which appears in the icons is simply an abbreviation of “LabVIEW to FrameL”. The “.vi” appearing after the code interface node names is the file extension for LabVIEW virtual instrument files.

²The acronym “UTC” is a compromise between the abbreviations of the English “Coordinated Universal Time” and the French “*Temps universel coordonné*”.

1980 00:00:00; since early 2002 GPS times specified in this way have been 9-digit numbers beginning with 7. OpenFile returns a raw pointer to the internal FrameH structure. At this point the frame is not associated with a particular file. The underlying subroutine

Listing 2.2: The OpenFrame code interface node subroutine

```
#include "extcode.h"
#include "hosttype.h"
#include "FrameL.h"
#include "FrVect.h"
#define frameErr 999
CIN MgErr CINRun(uInt32 *gpsStart, int32 *length, uInt32 *frame) {
    FrHistory* history = NULL;
    CStr ccomment = NULL;
```

creates the FrameH structure and initialises some of its members:

Listing 2.3: The OpenFrame code interface node subroutine (continued)

```
*frame = (uInt32) FrameHNew("ACIGA");
if (frame == NULL) {
    DbgPrintf("FrameHNew failed: %s\n", FrErrorGetHistory());
    return frameErr;
}
((FrameH*) *frame)->run = 0;
((FrameH*) *frame)->dataQuality = 0;
((FrameH*) *frame)->GTimeS = (unsigned int) *gpsStart;
((FrameH*) *frame)->GTimeN = 0;
((FrameH*) *frame)->ULeapS = FRGPSDELTA;
((FrameH*) *frame)->dt = *length;
```

It also adds a FrHistory structure identifying the originating program:

Listing 2.4: The OpenFrame code interface node subroutine (continued)

```
ccomment = DSNewPtr(SPrintf(NULL, (CStr) "FrameL %0.2f <- LabVIEW CIN",
    FrLibVersion(NULL)));
SPrintf(ccomment, (CStr) "FrameL %0.2f <- LabVIEW CIN", FrLibVersion(NULL));
history = FrHistoryAdd((FrameH*) *frame, (char*) ccomment);
if (history == NULL) {
    DbgPrintf("FrHistoryAdd failed: %s\n", FrErrorGetHistory());
    return frameErr;
}
DSDisposePtr(ccomment);
history->time = (unsigned int) *gpsStart;
return noErr;
}
```

The AddData code interface node takes a frame and adds a channel of raw data to it, returning the resultant frame. The data are stored as 16-bit signed integers scaled between -32,760 and +32,760, which is slightly less than the maximum possible range; input arguments specifying the expected maximum and minimum values of the data determine the scaling. Other input arguments specify the channel name and number, the length

of the data in seconds, a string indicating the physical units of the data, and optionally a down-sampling ratio (a value of 1 will perform no down-sampling). The underlying subroutine

Listing 2.5: The AddData code interface node subroutine

```
#include "extcode.h"
#include "hosttype.h"
#include "FrameL.h"
#include "FrVect.h"
#define frameErr 999
#define minshort (-32760.0)
#define maxshort (+32760.0)
typedef struct {
    int32 dimSize;
    float32 dataS1[1];
} TD1;
typedef TD1 **TD1Hdl;
CIN MgErr CINRun(ulnt32 *frame, LStrHandle chName, ulnt32 *chNumber, int32
    *length, float32 *min, float32 *max, LStrHandle units, TD1Hdl dataS, ulnt32
    *downSample) {
    FrAdcData* adc = NULL;
    CStr cchName = NULL, cunits = NULL;
    float bias = 0, slope = 0;
    double sampleRate = 0;
    FRLONG nData = 0;
    int i = 0, j = 0;
    float32 dataS1 = 0;
```

creates the FrAdcData structure, automatically computing the sample rate and the linear coefficients required to reconstruct the original scaling of the signal:

Listing 2.6: The AddData code interface node subroutine (continued)

```
cchName = DSNewPtr(SPrintf(NULL, (CStr) "K0:ANU-%H", chName));
SPrintf(cchName, (CStr) "K0:ANU-%H", chName);
cunits = DSNewPtr(SPrintf(NULL, (CStr) "%H", units));
SPrintf(cunits, (CStr) "%H", units);
nData = (FRLONG) floor(1.0 * (*dataS)->dimSize / *downSample);
sampleRate = (1.0 * nData) / (1.0 * (*length));
bias = ((*min) * maxshort - (*max) * minshort) / ((*min) - (*max));
slope = (minshort - maxshort) / ((*min) - (*max));
adc = FrAdcDataNewF((FrameH*) *frame, (char*) cchName, NULL, 0, (unsigned
    int) *chNumber, 16, (-bias / slope), (1.0 / slope), (char*) cunits,
    sampleRate, nData);
if (adc == NULL) {
    DbgPrintf("FrAdcDataNewF failed: %s\n", FrErrorGetHistory());
    return frameErr;
}
DSDisposePtr(cchName);
DSDisposePtr(cunits);
```

It then scales and writes the data from the input argument to the FrAdcData structure, down-sampling by the given ratio by simply averaging together blocks of data points:

Listing 2.7: The AddData code interface node subroutine (continued)

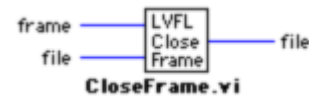
```

for (i = 0; i < nData; i++) {
    dataS1 = 0;
    for (j = 0; j < *downSample; j++) {
        dataS1 = dataS1 + (*dataS)->dataS1[((*downSample) * i) + j];
    }
    dataS1 = dataS1 / *downSample;
    dataS1 = (slope * dataS1) + bias;
    adc->data->dataS[i] = (short) dataS1;
}
return noErr;
}

```

The above down-sampling algorithm is not very sophisticated: ideally it should incorporate anti-alias filtering to prevent the signal from being corrupted by higher-frequency content. It is meant only to be used to down-sample signals where the highest frequencies are much lower than the Nyquist frequency, so that higher-frequency content can safely be assumed to be zero (see the discussion on page 27); it is convenient to perform the down-sampling immediately to reduce the amount of data needing to be handled.

The CloseFrame code interface node takes the given frame, writes it to the given Frame file and closes the frame. This may also close the file internally, if it was full, and open a new file. This is accomplished by the underlying subroutine:



Listing 2.8: The CloseFrame code interface node subroutine

```

#include "extcode.h"
#include "hosttype.h"
#include "FrameL.h"
#include "FrVect.h"
#define frameErr 999
CIN MgErr CINRun(ulnt32 *file, ulnt32 *frame) {
    if (FrameWrite((FrameH*) *frame, (FrFile*) *file) != NULL) {
        DbgPrintf("FrameWrite failed: %s\n", FrErrorGetHistory());
        return frameErr;
    }
    FrameFree((FrameH*) *frame);
    *frame = NULL;
    return noErr;
}

```

The CloseFile code interface node closes a Frame file. Its underlying subroutine is:



Listing 2.9: The CloseFile code interface node subroutine

```

#include "extcode.h"

```

```

#include "hosttype.h"
#include "FrameL.h"
#include "FrVect.h"
#define frameErr 999
CIN MgErr CINRun(uInt32 *file) {
    FrFileOEnd((FrFile*) *file);
    *file = NULL;
    return noErr;
}













```

2.2.4 Virtual circuitry

LabVIEW's visual interface makes it easy to write simple programs quickly, and it provides a wealth of built-in functionality targeted at data acquisition and experimental control. Programming more complicated behaviour can however lead to difficulties. For instance, LabVIEW executes its programs by checking for dependencies between components, as indicated by wires, and executing them as required. This means that the precise order of execution of components is not always self-evident, and this can lead to unexpected and undesired behaviour unless given greater attention. The visual nature of LabVIEW programming can also sometimes be a hindrance, as an unstructured layout of components, while not affecting the program's execution, can make understanding behaviour and effecting modifications more difficult.

The code interface nodes were used to substantially modify the original LabVIEW data acquisition software. In doing so it was found to be necessary to compile the FrameL library as a shared library, as opposed to a static library. A static library is simply a collection of pre-compiled source code which is embedded into programs at compile time; each program therefore having its own distinct copy of the library. However the subroutines perform actions which must be recognised by every other subroutine; for example, opening a Frame file creates an internal structure in the FrameL library which must be available to every other subroutine, if they are to refer to the same open file. A shared library, on the other hand, is an independent code object which is linked to programs at compile time; each program then accesses the same copy of the library when executed, producing the desired behaviour.

Images of the LabVIEW data acquisition software diagram and front panel are shown in figures 2.2 and 2.3 respectively. It will aid the understanding of LabVIEW software diagrams to be familiar with a few basic components:

- Components are joined by wires representing e.g. integers , booleans  and arrays of single precision real numbers .
- Variables are set by controls on the front panel. They can represent 16-bit unsigned integers  and booleans ; their icons usually have thick borders.
- Variables can be displayed on the front panel by indicators. They can represent 32-bit signed integers , booleans , strings  and floating point number arrays ; their icons usually have thin borders.
- Constants can represent numbers , booleans  and strings .

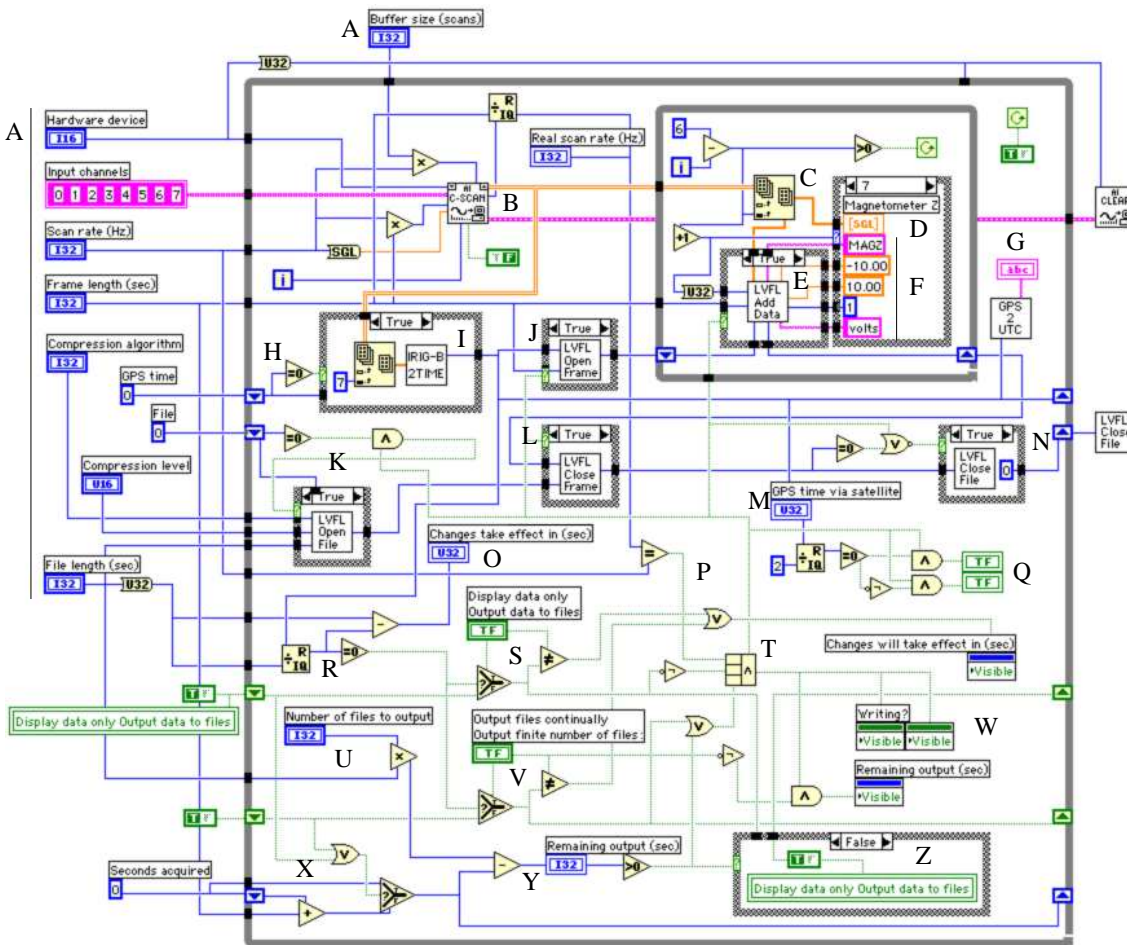



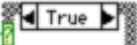










Figure 2.2: The LabVIEW data acquisition software diagram

- Operators can perform a variety of numeric , logical  and relational  operations.
- Case structures make decisions based upon boolean  or integer  input. Data can also be passed through the sides of the case structure .
- Loop structures  increment a counter  until the test  becomes false. Data passed through the sides of the loop structure  are read on the first iteration; data can also be passed from iteration to iteration using shift registers .
- Additional properties of controls and indicators can be read and modified: .

The upper half of the diagram (figure 2.2) is mostly concerned with acquiring, writing and storing data. Controls on the front panel (A, figures 2.2 and 2.2) specify the hardware device to acquire data from (in this case, a number representing the data acquisition card), the rate at which to acquire data, the size of a buffer to store acquired data, any data compression algorithms to use, and the number of seconds to store per frame and per Frame file. These settings are read at startup into a large infinite loop. A hardware acquisition component (B, figure 2.2) acquires data from the data acquisition card and produces a table, the columns of which contain data from the three seismometers, the

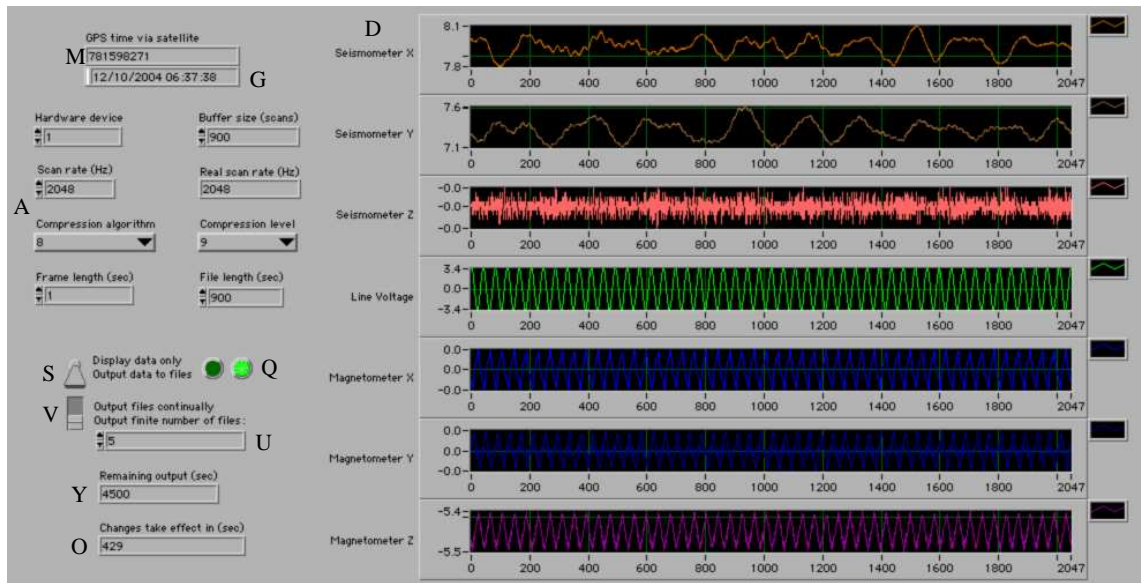


Figure 2.3: The LabVIEW data acquisition software front panel

three axes of the magnetic field sensor, the mains voltage monitor and from the GPS time receiver. The table is sent to an inner loop, where an array component (C) extracts each column in turn and sends then to a graph control on the front panel (D, figures 2.2 and 2.3) and to an AddData code interface node (E, figure 2.2).

The table column of the GPS time receiver is extracted by another array component and sent to an IRIG-B interpreter (I). This component interprets the waveform produced by the GPS time receiver and returns a GPS time, specified as the number of seconds since UTC January 6th 1980 00:00:00. Unfortunately the GPS time thus produced was found to be unstable, having a tendency to jump by random intervals intermittently over a long period of time, and occasionally produced a completely mangled signal when starting the data acquisition software. The reason for this instability is unknown but is probably some combination of loss of quality in the reception of the GPS signal via satellite, deficiencies in the GPS time receiver itself, and lack of robustness of the interpreter. This problem was circumvented by using the GPS time produced by the interpreter only on the first iteration, as decided by H; thereafter the time standard is maintained effectively by the synthetic function generator, whose manufacturer specifies an accuracy of 5 parts per million at 20–30 °C, degenerating at 5 parts per million per year [34]. The GPS time being recorded is displayed on the front control panel both as a raw integer (M, figures 2.2 and 2.3) and as a human-readable date (G); it also drives a pair of retro blinking lights (Q).

The GPS time is sent to an OpenFrame code interface node (J, figure 2.2) to create a frame, which is then sent via a shift register to the inner loop, where data is iteratively added to the frame by E, using the parameters supplied by F. Frame files are opened as required by K and are sent with the completed frame to L, where the frame is written to the Frame file. The Frame file is then closed as required by N.

The lower half of the diagram is mostly concerned with the user interface and control of data acquisition. One made made of the GPS time stamping, imposed by the components R, was the the starting GPS time for the initial frame in a Frame file, i.e. the starting GPS time for the Frame file, must be a multiple of the total length in seconds of the Frame

file. This will mean that, for example, Frame files of 15-minute lengths must start either on the hour or at 15, 30 or 45 minutes past it. While not important for the data itself, this condition greatly facilitates the organisation and packaging of very large numbers of Frame files.

Two key controls are the boolean switches **S** and **V** (figures 2.2 and 2.3). The former specifies whether or not the software should write acquired data to files; the latter specifies whether the software should write files continually or only write a finite number of files. In this latter case, the number of files to acquire and write is specified by **U**, and the components **X** (figure 2.2) calculate the number of seconds of data acquired, and display the number of seconds remaining to be acquired (**Y**, figures 2.2 and 2.3). When all files have been acquired, the software switches itself off through the control **Z** (figure 2.2).

The values of the switches **S** and **V** are maintained in shift registers, which are only allowed to take the switches' current values when the output of **R** is true; the indicator **O** shows the time to elapse before the switches' current values will actually take effect. When this does happen, the logical operator **T** controls whether data is written to files: it controls the boolean case structures surrounding **K**, **J**, **E**, **L** and **N**, and it controls the visibility of some controls and indicators (**W**).

Despite much effort to produce a reasonably sophisticated, flexible, and usable piece of software, at least one deficiency remains. It was discovered, some months after the software was thought to be complete, that the functionality to create frames of lengths not equal to 1 second had not been implemented correctly, although this functionality was never actually used. Nevertheless, the addition of the ability to write Frame files directly represents a substantial practical improvement to the data acquisition software.

2.2.5 Uploading files

Files acquired and written by the data acquisition software, requiring no further processing, are uploaded to the Mass Data Storage System using a freeware Macintosh FTP client called FTupperWare [41]. This software simply monitors a folder specified by the user, and uploads any files placed in that folder to a specified FTP server. The only remaining problem was to move the Frame files from the LabVIEW directory (where they were forced to be written due to the problem, discussed on page 10, concerning using path names with the FrameL library) to the FTupperWare hot folder. This was accomplished using a short AppleScript:

Listing 2.10: The MoveFiles AppleScript

```
tell application "Finder"
  delete (every file of folder "LabVIEW 5.1" of startup disk whose name contains
    ".gwf" and name contains "_OPEN")
  repeat
    move (every file of folder "LabVIEW 5.1" of startup disk whose name contains
      ".gwf" and name does not contain "_OPEN") to folder "Hot Folder"
      of folder "FTupperWare" of startup disk
  end repeat
end tell
```

Files which are incomplete are distinguished by the "_OPEN" suffix. After deleting any old incomplete files, the script moves all completed Frame files to the FTupperWare hot folder.

2.3 The data storage system

Data from the physical environment monitoring station has been stored on the Mass Data Storage System since 2002, although the record is far from continuous (see figure 2.4). This section details the original format of the data and subsequent reprocessing.

2.3.1 The original format

The data was originally stored in 28,246 uncompressed Frame files with an average size of 13 megabytes, each file containing 1000 frames of length 1 second. In total the files consumed 345 gigabytes.

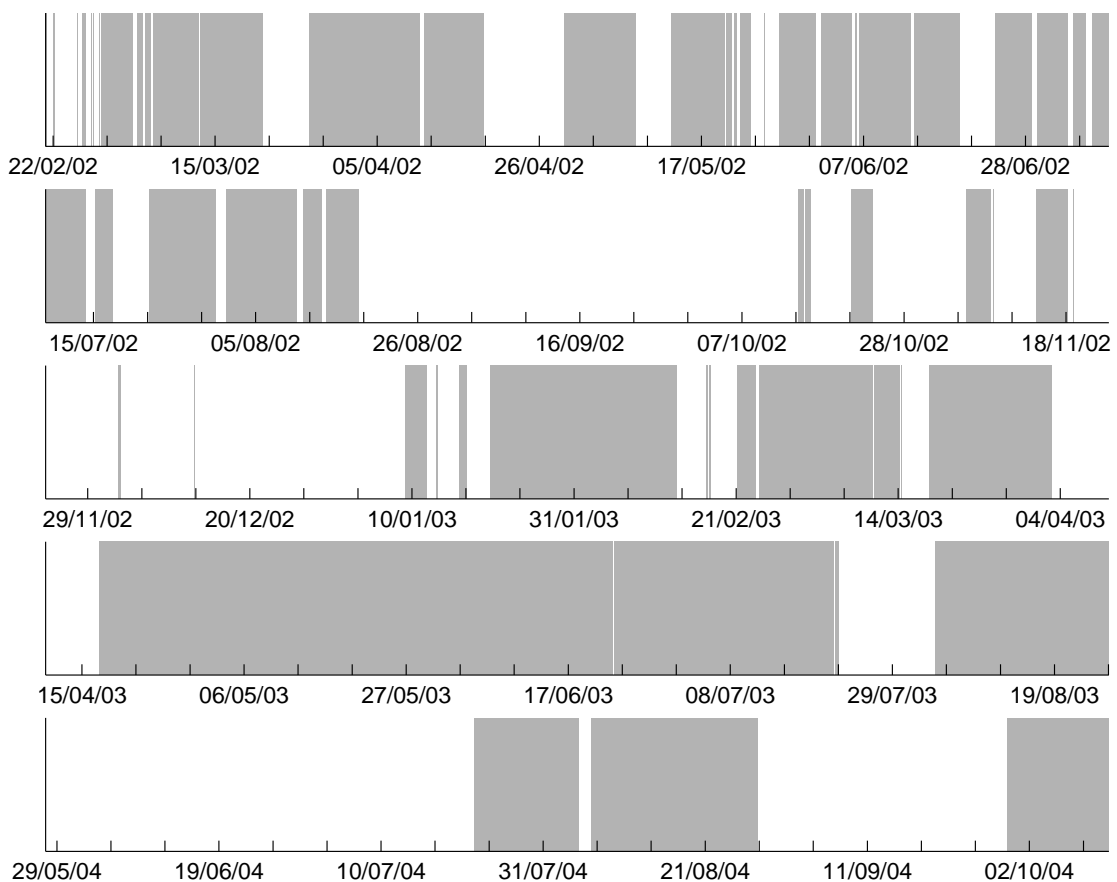


Figure 2.4: Timeline of the physical environment monitoring station data; the shaded areas indicate the times for which data exists in the Mass Data Storage System.

2.3.2 Reprocessing

Each Frame file was compressed using the FrameL library's internal compression algorithms. Some modifications were made to the channel names to make them compliant with the formal naming standard [38]. The Frame files were then packaged into `.tar.gz` compressed archive files, such that each archive contained continuous data. The result was 1,658 files with an average size of 127 megabytes, consuming a total of 205 gigabytes. Although in theory the data now consumes 40% less storage space, due to the nature of

tape-based file systems that space is likely to be unrecoverable. Nevertheless the vastly reduced number of files makes the data set much easier to manage. Due to the large size of the data set, scripts were written to download the files from the storage system to an attached scratch disk, perform the reprocessing and then upload the files back to the storage system, a process which took several days to complete.

2.4 Data acquisition in practice

This section recounts the practicalities of operating the local physical environment monitoring station, and of acquiring data from the US LIGO Observatory and the French-Italian VIRGO Project.

2.4.1 Locally

The physical environment monitoring station was brought into operation, for the first time after the modifications described in sections 2.1 and 2.2, on July 22nd 2004 (see figure 2.4). It operated until August 4th, when it was shut down in order to disconnect and send the faulty vertical (*Z*) seismometer for repair. The station was restarted on August 6th, and was operational until crashing on August 27th. It was found that the freeware FTP client software FTupperWare was for some reason no longer uploading the files being moved into its hot folder, which eventually caused the acquisition software to consume all 6 gigabytes of the Macintosh's hard drive. At the time it was thought that this may have been caused by a momentary network outage or the Mass Data Storage System temporarily being unavailable.

The station was restarted on September 2nd, but subsequently crashed again on September 18th, in very similar circumstances to August 27th. It is now believed that the crashes were caused by the Macintosh operating system MacOS 9.2.2 and its lack of pre-emptive multi-tasking, a feature of most modern operating systems which allows multiple running programs to more equally share processor time. It is likely that FTupperWare was effectively being suspended by the processor-intensive data acquisition software, and was therefore unable to upload the files. It was also discovered that the data acquired during this period had not been properly time-stamped, for reasons unknown, and was thus deleted. The station was restarted again on September 29th and operated without further incident until October 12th and the conclusion of data acquisition for this project.

While the station's modified LabVIEW data acquisition software performed well under the circumstances, the station will be moved onto a Windows PC in the near future.

2.4.2 From LIGO

Data was acquired from LIGO using the LIGOTOOLS software utility package [42]. The utility `getFrames` was used to acquire data, stored on the central server at the California Institute of Technology, from the LIGO S2 Science Run, which was in progress from February 9th to April 14th 2003.

More recent data was acquired in real time directly from the observatories at Hanford and Livingston. This acquisition was set up by Antony Searle of the Department of Physics, ANU, using a small C++ [43] program which, when executed, would instruct `getFrames` to acquire 1 hour of data, starting from 24 hours in the past. The program was automatically executed every hour using a UNIX cron job.

2.4.3 From VIRGO

Acquisition of recent data from VIRGO was carried out on our behalf by Benoit Mours of the Laboratoire d'Annecy-le-vieux de Physique des Particules, France, who packaged the data required into Frame files and uploaded them to a local server.

Data analyses

The theory of data analyses is for the most part heuristic in character. The importance of the derivation and mathematical properties of an analysis are usually secondary; of primary importance is whether the analysis was successful in carrying out the tasks required of it, and in particular whether it found anything of physical interest. The consequence is that it is often the case that there is no generally agreed upon method of performing certain tasks, rather a sometimes overwhelming multitude of different ideas, each with their unique features and flaws. One is free to experiment with these ideas, however, and with one's own, in the hope that the particular combination arrived at might show a new way forward.

This chapter presents the data analyses that will be utilised in analysing the data from the physical environment monitors. Some analyses were chosen because they are very widely used, and therefore their implementation and manner of interpretation are well established. Others were cobbled together from various ideas where there were no well-established procedures. A few were chosen simply because, in the author's opinion, they were interesting and appeared worthy of further investigation.

3.1 Long timescale analyses

This section focuses on analyses designed to find features that might present themselves over long periods of time.

3.1.1 The autocorrelation

The autocorrelation $\mathcal{A}(\tau)$ of a time series X measures to what extent oscillations in X at time t are reproduced at time $t + \tau$, over the duration of the time series. It is useful for finding persistent recurring features of the time series. For a continuous time series $X(t)$, the autocorrelation is

$$\mathcal{A}[X(t)](\tau) = \int_{-\infty}^{\infty} X(t)X(t + \tau) dt. \quad (3.1)$$

For a discrete time series X_t of length n , the autocorrelation is

$$\mathcal{A}[X_t]_{\tau} = \sum_{t=0}^{n-\tau-1} X_t X_{t+\tau}, \quad (3.2)$$

where

$$\mathcal{A}[X_t]_{-\tau} = \mathcal{A}[X_t]_{\tau}. \quad (3.3)$$

From numerical experimentation using MATLAB, the following observations were made. An autocorrelation of a random time series was found to be flat, apart from a large spike at $\tau = 0$ where the series exactly matches itself. An autocorrelation of a time series where oscillations reproduced themselves in the short term were found to have a few oscillations either side of the central peak before reducing to a flat background. Autocorrelations of periodic time series were found to be periodic with the same period.

3.1.2 The correlation

The correlation $\mathcal{C}(\tau)$ of two time series X and Y measures to what extent oscillations in X at time t are reproduced in Y at time $t + \tau$, over the duration of the time series. It is useful for finding persistent and possibly causally related features present in both time series. For continuous time series $X(t)$ and $Y(t)$ the correlation is

$$\mathcal{C}[X(t), Y(t)](\tau) = \int_{-\infty}^{\infty} X(t)Y(t + \tau) dt. \quad (3.4)$$

For discrete time series X_t and Y_t of length n , the correlation is

$$\mathcal{C}[X_t, Y_t]_{\tau} = \sum_{t=\begin{cases} 0, \tau \geq 0 \\ \tau, \tau < 0 \end{cases}}^{\begin{cases} n-\tau-1, \tau \geq 0 \\ n-1, \tau < 0 \end{cases}} X_t Y_{t+\tau}. \quad (3.5)$$

Numerical experimentation within MATLAB resulted in some observations. Correlations of two random time series were found to be random. Correlations of two time series which were both polynomials of a single random time series were found to be flat with a sharp central peak. Correlations of two sinusoidal time series were found to be periodic, but also asymmetric in the amplitude and period of the oscillations about the $\tau = 0$ axis, the degree of asymmetry being proportional to the difference between the periods of the sinusoidal time series. Two time series where a distinct oscillation, appearing in X at time t , was reproduced in Y at time $t + \tau$, which might signify a causal influence propagating from X to Y , produced an asymmetric correlation, with a feature distinguishable from the rest of the correlation appearing at $\mathcal{C}(-\tau)$ but not at $\mathcal{C}(\tau)$.

3.1.3 The power spectrum

A power spectrum $\mathcal{P}(f)$ of a time series X measures the relative strength of those oscillations in X that are of frequency f . There are several different ways of defining a power spectrum [44]; by far the most common one uses the Fourier transform. A Fourier transform is a relation \mathcal{F} between two (in general) complex functions $X(t)$ and $\Phi(f)$ such that

$$\Phi(f) = \mathcal{F}[X(t)](f) = \int_{-\infty}^{\infty} X(t)e^{-2\pi ift} dt \quad (3.6)$$

and

$$X(t) = \mathcal{F}^{-1}[\Phi(f)](t) = \int_{-\infty}^{\infty} \Phi(f) e^{+2\pi i f t} df. \quad (3.7)$$

There are many slightly different definitions of the transform. They differ from each other in details such as, for example: the names t and f are often replaced by x and p , and f is often replaced by $\omega = 2\pi f$; the factor of 2π in the exponent may be dropped, and there may exist a normalising factor of 2π or $\sqrt{2\pi}$; and the $+$ and $-$ signs in the exponents may be interchanged. These differences reflect the wide use of the Fourier transform in pure and applied mathematics, classical and modern physics, engineering, and signal processing [45], the last of which uses the above convention of equations 3.6 and 3.7. The power spectrum is defined to be the complex modulus squared of the Fourier transform of the time series:

$$\mathcal{P}[X(t)](f) = |\Phi(f)|^2 = \left| \int_{-\infty}^{\infty} X(t) e^{-2\pi i f t} dt \right|^2. \quad (3.8)$$

In the case of a real time series X , the Fourier transform and the power spectrum have some important properties. We observe that

$$\Phi(f) = \int_{-\infty}^{\infty} X(t) e^{-2\pi i f t} dt \quad (3.9)$$

$$= \int_{-\infty}^{\infty} X(t)^* e^{-2\pi i f t} dt \quad (3.10)$$

$$= \left[\int_{-\infty}^{\infty} X(t) e^{-2\pi i (-f) t} dt \right]^* \quad (3.11)$$

$$= \Phi(-f)^*. \quad (3.12)$$

This reflective symmetry implies that only half (e.g. $f > 0$) of the Fourier transform is storing unique information, since the other half (i.e. $f < 0$) is merely its mirror image. It follows that

$$\mathcal{P}(f) = |\Phi(f)|^2 = \Phi(f)^* \Phi(f) = \Phi(-f) \Phi(-f)^* = \mathcal{P}(-f). \quad (3.13)$$

This implies that the Fourier power spectrum does not distinguish between f and $-f$. This phenomenon is referred to as aliasing; f and $-f$ are said to be aliases of each other [46].

For a discrete time series X_t of length n , the discrete Fourier transform may be derived from the continuous transform in two steps. The first step replaces continuous time with discrete time points spaced at intervals of width Δ :

$$t \rightarrow t\Delta, \quad (3.14)$$

$$\int_{-\infty}^{\infty} X(t) e^{-2\pi i f t} dt \rightarrow \sum_{t=0}^{n-1} X_t e^{-2\pi i f t \Delta} \Delta, \quad (3.15)$$

$$\int_{-\infty}^{\infty} \Phi(f) e^{+2\pi i f t} df \rightarrow \int_{-\infty}^{\infty} \Phi(f) e^{+2\pi i f t \Delta} df. \quad (3.16)$$

This first step has the very important consequence of introducing additional aliasing into

the discrete Fourier transform and power spectrum. We observe that

$$\Phi\left(f + \frac{1}{\Delta}\right) = \sum_{t=0}^{n-1} X_t e^{-2\pi i(f + \frac{1}{\Delta})t\Delta} \Delta \quad (3.17)$$

$$= \sum_{t=0}^{n-1} X_t e^{-2\pi i(ft\Delta + t)} \Delta \quad (3.18)$$

$$= \sum_{t=0}^{n-1} X_t e^{-2\pi ift\Delta} e^{-2\pi it} \Delta \quad (3.19)$$

$$= \sum_{t=0}^{n-1} X_t e^{-2\pi ift\Delta} \Delta \quad (3.20)$$

$$= \Phi(f). \quad (3.21)$$

This periodicity on its own would imply that any segment of length $1/\Delta$ will store all the unique information present in the discrete Fourier transform. The addition of the reflective aliasing described above, however, reduces this length by a factor of 2; the segment must also be symmetric about the $f = 0$ axis. This implies that the segment $[0, 1/2\Delta]$ will store all the unique information present in the discrete Fourier transform, and hence also in the power spectrum. Therefore, the maximum possible frequency representable by a discrete Fourier transform is one half of the sampling frequency: $\frac{1}{2\Delta}$. This is the *sampling theorem*; $1/2\Delta$ is known as the Nyquist frequency [47].

The second and final step in the derivation of the discrete Fourier transform from the continuous transform replaces continuous frequency with n discrete frequency points spaced at $1/n\Delta$ intervals between 0 and $1/\Delta$:

$$f \rightarrow \frac{f}{n\Delta}, \quad (3.22)$$

$$\sum_{t=0}^{n-1} X_t e^{-2\pi ift\Delta} \Delta \rightarrow \sum_{t=0}^{n-1} X_t e^{-2\pi i\frac{f}{n}t} \Delta, \quad (3.23)$$

$$\int_{-\infty}^{\infty} \Phi(f) e^{+2\pi ift\Delta} df \rightarrow \sum_{f=0}^{n-1} \Phi_f e^{+2\pi i\frac{f}{n}t} \frac{1}{n\Delta}. \quad (3.24)$$

The factor of Δ can be cancelled out, leading to the complete definition of the discrete Fourier transform:

$$\Phi_f = \mathcal{F}[X_t]_f = \sum_{t=0}^{n-1} X_t e^{-2\pi ift/n}, \quad (3.25)$$

$$X_t = \mathcal{F}^{-1}[\Phi_f]_t = \frac{1}{n} \sum_{f=0}^{n-1} \Phi_f e^{+2\pi ift/n}. \quad (3.26)$$

The normalising factor $1/n$ is sometimes discarded [48]. The discrete power spectrum is defined to be:

$$\mathcal{P}[X_t]_f = |\Phi_f|^2 = \left| \sum_{t=0}^{n-1} X_t e^{-2\pi ift/n} \right|^2. \quad (3.27)$$

A practical consequence of changing from continuous to discrete frequency is that f

now represents not a single frequency but a bin of frequencies centred on f . It turns out that these bins are not sharply delineated from each other, but instead leak into each other to a significant extent. This results in the power spectrum estimation of the power at frequency f being corrupted by “leakage” from power estimates at neighbouring frequencies. This effect can be reduced by pre-multiplying the time series by a window function. The ideal window function should have a narrow peak in the centre of the time series but fall off rapidly to zero at either side, but in fact these two properties are mutually exclusive. Real window functions are always one particular trade-off between these two properties [44].

It is important to realise that, regardless of the frequency content of the time series X , the power spectrum can only compute the power of frequencies up to one half of the sampling frequency. Any time series content at higher frequencies will be aliased into the power spectrum, giving a spurious estimation of the power at some lower frequency [44]. To avoid this it is important to select a sampling frequency that adequately covers the range of frequencies expected to appear in the time series. Additionally, anti-aliasing filters can be used (see page 9) to filter out any higher frequency content prior to computing the power spectrum.

When computing the power spectrum of a time series over a long period of time, it is better to break the time series into smaller blocks, compute the power spectrum \mathcal{P}_f^b of each block, then add them together to arrive at an averaged power spectrum $\tilde{\mathcal{P}}_f = \sum_b \mathcal{P}_f^b$. This approach can still result in good frequency resolution, while providing much lower variance in the power estimates and at lower computational cost. In some cases better results can also be obtained by partially overlapping the blocks [44].

3.1.4 The cross spectrum and the coherence

The cross spectrum \mathcal{X} of two time series X and Y measures the combined relative strength and the phase difference of oscillations in X and Y that are of frequency f . In the context of Fourier transforms it is defined to be

$$\mathcal{X}[X(t), Y(t)](f) = \mathcal{F}[X(t)](f)^* \mathcal{F}[Y(t)](f). \quad (3.28)$$

The cross spectrum is, in general, complex-valued. It is related to the power spectrum by

$$\mathcal{X}[X(t), X(t)](f) = \mathcal{P}[X(t)](f). \quad (3.29)$$

It is also related to the correlation by the correlation theorem [44]:

$$\mathcal{X}[X(t), Y(t)](f) = \mathcal{F}[\mathcal{C}[X(t), Y(t)](\tau)](f). \quad (3.30)$$

A more useful quantity derived from the cross spectrum is the coherence \mathcal{H} , defined for two time series X and Y as the complex modulus squared of the averaged cross spectrum of X and Y divided by the averaged power spectra of X and of Y :

$$\mathcal{H}[X(t), Y(t)](f) = \frac{|\tilde{\mathcal{X}}[X(t), Y(t)](f)|^2}{\tilde{\mathcal{P}}[X(t)](f) \tilde{\mathcal{P}}[Y(t)](f)} \quad (3.31)$$

$$= \frac{|\sum_b \mathcal{X}^b[X(t), Y(t)](f)|^2}{\sum_b \mathcal{P}^b[X(t)](f) \sum_b \mathcal{P}^b[Y(t)](f)}, \quad (3.32)$$

where averaging is performed over small blocks of the time series. The coherence is inherently an averaged quantity; it cannot be computed using non-averaged quantities:

$$\mathcal{H}^{\text{non-averaged}} = \frac{\left| \mathcal{X}[X(t), Y(t)](f) \right|^2}{\mathcal{P}[X(t)](f) \mathcal{P}[Y(t)](f)} \quad (3.33)$$

$$= \frac{\left| \mathcal{F}[X(t)](f)^* \mathcal{F}[Y(t)](f) \right|^2}{\mathcal{P}[X(t)](f) \mathcal{P}[Y(t)](f)} \quad (3.34)$$

$$= \frac{\left(\mathcal{F}[X(t)](f)^* \mathcal{F}[Y(t)](f) \right)^* \left(\mathcal{F}[X(t)](f)^* \mathcal{F}[Y(t)](f) \right)}{\mathcal{F}[X(t)](f)^* \mathcal{F}[X(t)](f) \mathcal{F}[Y(t)](f)^* \mathcal{F}[Y(t)](f)} \quad (3.35)$$

$$\equiv 1, \quad (3.36)$$

whereas

$$\mathcal{H} = \frac{\left| \tilde{\mathcal{X}}[X(t), Y(t)](f) \right|^2}{\tilde{\mathcal{P}}[X(t)](f) \tilde{\mathcal{P}}[Y(t)](f)} \quad (3.37)$$

$$= \frac{\left| \mathcal{X}^1[X(t), Y(t)](f) + \mathcal{X}^2[X(t), Y(t)](f) + \dots \right|^2}{\left(\mathcal{P}^1[X(t)](f) + \mathcal{P}^2[X(t)](f) + \dots \right) \left(\mathcal{P}^1[Y(t)](f) + \mathcal{P}^2[Y(t)](f) + \dots \right)} \quad (3.38)$$

is a function between 0 and 1. A value of 0 indicates no dependence between X and Y , while a value of 1 would imply complete dependence [46]. In general, the significance of a particular peak in a graph of coherence should only be judged relative to the noise floor.

3.2 Short timescale analyses

This section focuses on analyses designed to search for transient events that would only be present over very short timescales.

3.2.1 What is a transient event?

In order to proceed we must first define what is meant by a transient event. Unfortunately, this is in fact the fundamental problem with such analyses. While long timescale correlations are, by their very nature, stable over a long period of time, and hence more easily defined and identified, short timescale events are much more difficult to characterise. Their definition may therefore conceivably encompass any number of transitory phenomena that may or may not be significant.

The precise definition of significance in itself brings further difficulties. The conventional approach is to devise an algorithm to compute a numerical test statistic at various times, then impose a threshold. Only computed values of the test statistic greater than the threshold are considered to imply significant events. This requires substantial previous experience of the data, however, in order to set an appropriate threshold, which may even then exclude events which should have in fact been considered significant. The only conclusion is that the notion of significance is too vague a concept to be encoded in a general algorithm. The best that can be achieved is to calculate multiple test statistics, each encoding one possible way in which a transient event might be distinguished. Ultimately

it is the observing scientists who must interpret the results of their analyses and decide whether or not they constitute anything significant.

The remaining section details the development of the transient event search algorithm and the statistical tests that were used.

3.2.2 The power spectrogram and the cross spectrogram

The power spectrogram $\mathcal{S}^{\mathcal{P}}$ of a time series X attempts to estimate, using the power spectrum, the instantaneous power at time t of oscillations in X that are of frequency f . Technically the notion of instantaneous power is undefined, since the power spectrum of a time series can only be computed after observing the time series for a non-zero interval time, thereby introducing uncertainty as to when exactly during that time interval the time series possessed the computed power spectrum. Therefore a more accurate definition is that the power spectrogram computes the power spectrum of a window located about time t . The size and nature of this window are determined by its function W , which is translated along the time series X as a function of t . The complete definition is:

$$\mathcal{S}^{\mathcal{P}}[X(\tau)](f, t) = \mathcal{P}[X(\tau)W(\tau - t)](f). \quad (3.39)$$

The cross spectrum $\mathcal{S}^{\mathcal{X}}$ generalises the power spectrogram to two time series X and Y :

$$\mathcal{S}^{\mathcal{X}}[X(\tau), Y(\tau)](f, t) = \mathcal{X}[X(\tau)W(\tau - t), Y(\tau)W(\tau - t)](f). \quad (3.40)$$

The discrete forms of the spectrograms are formulated analogously:

$$\mathcal{S}^{\mathcal{P}}[X_{\tau}]_{f,t} = \mathcal{P}[X_{\tau}W_{\tau-t}]_f, \quad (3.41)$$

$$\mathcal{S}^{\mathcal{X}}[X_{\tau}, Y_{\tau}]_{f,t} = \mathcal{X}[X_{\tau}W_{\tau-t}, Y_{\tau}W_{\tau-t}]_f. \quad (3.42)$$

3.2.3 The search algorithm

The search algorithm takes as its basic premise that a transient event is any region of a spectrogram which is significantly different from its surroundings. It uses a discrete spectrogram $\mathcal{S}_{f,t}$ together with a square window function of length T :

$$W_t = \begin{cases} 1, & 0 \leq t < T; \\ 0, & \text{otherwise.} \end{cases} \quad (3.43)$$

The discrete spectrogram is divided by frequency and by time into bins of dimensions F, T , originating at the point (f, t) and containing an equal number of points:

$$\mathbf{B}_{f,t} = \{\mathcal{S}_{f',t'} \mid 0 \leq f' - f < F, 0 \leq t' - t < T\}. \quad (3.44)$$

Each bin $\mathbf{B}_{f,t}$ is then compared with bins displaced forward and backward in time using a test statistic $\mathcal{T}(\mathbf{B}_{f,t}, \mathbf{B}_{f,t+dt})$, in order to ascertain how, according to this statistical test, each bin differs from its surroundings. The bins are compared up to a maximum search

range $dt = \pm dt_{\max}$, excluding $dt = 0$:

$$\begin{aligned}
& \mathcal{T}(\mathbf{B}_{f,t}, \mathbf{B}_{f,t-dt_{\max}}), \\
& \mathcal{T}(\mathbf{B}_{f,t}, \mathbf{B}_{f,t-dt_{\max}+1}), \\
& \quad \dots \\
& \mathcal{T}(\mathbf{B}_{f,t}, \mathbf{B}_{f,t-1}), \\
& \mathcal{T}(\mathbf{B}_{f,t}, \mathbf{B}_{f,t+1}), \\
& \quad \dots \\
& \mathcal{T}(\mathbf{B}_{f,t}, \mathbf{B}_{f,t+dt_{\max}-1}), \\
& \mathcal{T}(\mathbf{B}_{f,t}, \mathbf{B}_{f,t+dt_{\max}}).
\end{aligned} \tag{3.45}$$

A final event statistic $\mathcal{E}_{f,t}$ is computed for the bin $\mathbf{B}_{f,t}$ by taking a geometric average of the above comparisons:

$$\mathcal{E}_{f,t} = \sqrt{\sum_{dt=1}^{dt_{\max}} \left(\mathcal{T}(\mathbf{B}_{f,t}, \mathbf{B}_{f,t-dt})^2 + \mathcal{T}(\mathbf{B}_{f,t}, \mathbf{B}_{f,t+dt})^2 \right)}. \tag{3.46}$$

The event statistics are then sorted into descending numerical order, with the algorithm returning an appropriate proportion of the highest values.

Five statistical tests were selected for use by the search algorithm for this project, and are described in the following sections. They were chosen to try to represent a range of different approaches to testing for differences between two populations of numbers. It was not considered as to whether or not the assumptions made by the statistical tests were satisfied by the data supplied to them; ultimately only the findings of the search algorithm, and whether or not they could be deemed significant was considered important. Indeed, if a particular test statistic consistently found transient features that were deemed significant, this might indicate that the data was indeed satisfying the assumptions of the test statistic, in which case a valuable insight would be gained into the nature of the data.

3.2.4 The difference-in-mean test

This statistical test is one of the simplest tests one could devise for testing for a possible difference between two populations. It simply computes the difference in the mean of the two bins:

$$\mathcal{T}_{\text{dmean}}(\mathbf{B}^1, \mathbf{B}^2) = \left| \text{mean}\{\mathbf{B}^1\} - \text{mean}\{\mathbf{B}^2\} \right|. \tag{3.47}$$

It was included principally for comparison against the more complicated statistical tests, to see whether or not they were performing better than a more simplistic test.

3.2.5 The Student paired t-test

This is one of the most commonly employed statistical tests. The paired form calculates the significance of the difference between the means of two populations, making no assumptions about their variances, but assuming the populations to be normally distributed [49, 50].

It is computed thus:

$$\mathcal{T}_{t-t}(\mathbf{B}^1, \mathbf{B}^2) = \frac{\text{mean}\{\mathbf{B}^1\} - \text{mean}\{\mathbf{B}^2\}}{\sqrt{\sum_{i=0}^{n-1} \left((\mathbf{B}_i^1 - \text{mean}\{\mathbf{B}^1\}) - (\mathbf{B}_i^2 - \text{mean}\{\mathbf{B}^2\}) \right)^2}}. \quad (3.48)$$

Unlike the Student t-test, the following three statistical tests make no assumptions about the populations being tested; they are distribution-free [51]. A distinguishing feature of distribution-free statistical tests is that they are mathematically much simpler than their distribution-dependent analogues, relying only on basic probability theory and combinatorics.

3.2.6 The Wilcoxon signed rank test

This is one of many statistical tests based upon the method of randomisation. It tests whether the measured data set $\{\mathbf{B}_i^1 - \mathbf{B}_i^2 \mid 0 \leq i < n\}$ was significantly more likely to have occurred than any data set obtained from the measured data set by randomly negating some or all of its members [51]. It is computed by taking the data set $\{\mathbf{B}_i^1 - \mathbf{B}_i^2 \mid 0 \leq i < n\}$, sorting them, then adding together the ranks of those values either greater or less than zero, producing the two test statistics:

$$W_+ = \sum \left\{ j \mid \text{sorted}\{\mathbf{B}_i^1 - \mathbf{B}_i^2 \mid 0 \leq i < n\}_j > 0 \right\}, \quad (3.49)$$

$$W_- = \sum \left\{ j \mid \text{sorted}\{\mathbf{B}_i^1 - \mathbf{B}_i^2 \mid 0 \leq i < n\}_j < 0 \right\}. \quad (3.50)$$

The test statistic used was the maximum (i.e. the most significant) of W_+ and W_- :

$$\mathcal{T}_{W_{\pm}} = \max\{W_+, W_-\}. \quad (3.51)$$

3.2.7 The Kolmogorov-Smirnov test

This tests for any general overall difference between two populations [49]. It is computed by first calculating the cumulative distribution functions

$$D^k(x) = \sum \{\mathbf{B}_i^k \mid i \leq x\}, \quad (3.52)$$

then finding the absolute maximum difference between them:

$$\mathcal{T}_{K-S} = \left| \max\{D^1(x) - D^2(x) \mid 0 \leq x \leq n\} \right|. \quad (3.53)$$

3.2.8 The Wilcoxon-Mann-Whitney rank sum test

This is another test of very general differences between two populations [49]. It is computed by pooling the two populations \mathbf{B}^1 and \mathbf{B}^2 into a single population \mathbf{B} , sorting it, and then counting the number of times an element of \mathbf{B}^1 precedes an element of \mathbf{B}^2 [51]:

$$U = \sum_{\substack{i,j=0 \\ i < j}}^{n-1} \begin{cases} 1, & \mathbf{B}_i^1 < \mathbf{B}_j^2; \\ 0, & \text{otherwise.} \end{cases} \quad (3.54)$$

The test statistic used was the absolute deviation of U from its expectation value:

$$\mathcal{T}_{\text{WMW}} = |E[U] - U| = \left| \frac{n(n+1)}{4} - \sum_{\substack{i,j=0 \\ i < j}}^{n-1} \begin{cases} 1, & \mathbf{B}_i^1 < \mathbf{B}_j^2; \\ 0, & \text{otherwise.} \end{cases} \right| \quad (3.55)$$

Data processing

A physical environment monitoring station, such as that described in section 2.1, outputs approximately 9 thousand data points each second it is operational. In a minute it will output 537 thousand data points; in a day, 774 million. Considering several physical environment stations over a period of weeks, the total output easily amounts to billions of data points. Performing the above-described analyses on this amount of data using a single computer within a reasonable time frame would be impossible; instead we must utilise the power of several computers, executing in parallel.

4.1 Some technologies

This section details some technologies in computer programming, in particular programming for parallel computers. They will be later used in the implementation of a parallel data analysis program.

4.1.1 Parallel programs

A process is an abstraction representing a single unit of an executing computer program. There are several different paradigms for how multiple processes can work in parallel. They differ from each other in details such as whether the parallel behaviour is defined implicitly by the compiler or explicitly by the programmer, whether processes have access to shared memory, whether processes can access each other's local memory, whether processes send messages between each other, and whether communication requires all processes to explicitly participate. Many of these paradigms were motivated by a particular computer architecture, such as e.g. vector-parallel processors [52].

Parallel programs generally refer to those in which the same program is executed on multiple processors, each generating a single process. The processes, initially identical, are able to uniquely distinguish themselves and thus modify their behaviour to perform different tasks. The processes may also be able to access shared memory and/or communicate by sending messages between each other.

4.1.2 Message-passing

In the message-passing parallel computing paradigm, processes have access only to their local memory, but communicate with each other by sending and receiving messages over a network. Both the sending and receiving process must explicitly participate; a process cannot send a message until the receiving process has also asked to receive the message. The message-passing paradigm does not specify the specific architecture of the network,

which could be the internal architecture of a supercomputer or cables connecting clusters of low-cost PCs. This makes message-passing a universal paradigm which can be implemented for, and allow programs written for it to run on, a wide range of high-performance and low-cost parallel computers. Another specific advantage is that it lends itself to compiler- and hardware-optimised memory management, which for most modern computer architectures is the key to high performance [52].

4.1.3 MPI

The Message-Passing Interface (MPI) [53] is the industry standard for implementations of the message-passing paradigm. The standard defines a number of key concepts [52, 54]:

- A communicator is a collection of processes which may communicate with each other. Each process is assigned a unique integer, called its rank, which it may use to distinguish itself from other processes in the communicator. A process may belong to more than one communicator; this allows processes to be partitioned into groups for the purpose of performing specific tasks.
- Message data are defined by the parameters (*address*, *count*, *datatype*). The *datatype* parameter specifies the type of data being sent; it may be a type elementary to the programming language being used, or it may be a custom type constructed by the programmer from within MPI. Specifying the data type independently means that an MPI implementation can ensure that the data is sent correctly irrespective of computer architecture or programming language differences. The parameters *count* and *address* are the number of instances of the data type being sent and the address of the first instance respectively.
- Messages to be sent are defined by the parameters (*address*, *count*, *datatype*, *dest*, *tag*, *comm*). In addition to the message data parameters, *comm* is the communicator on which the message will be sent, and *dest* is the rank of the process within the given communicator to which the message will be sent. The parameter *tag* optionally specifies a number with which to identify the message.
- Messages that are to be received are defined by the parameters (*address*, *count*, *datatype*, *source*, *tag*, *comm*). The message data parameters must be the same as the message parameters specified when the message was sent. In addition, *comm* is the communicator on which the message is to be received, and *source* is the rank of the process within the given communicator from which the message will be received. The parameter *tag* can be used to receive only messages with the same *tag*.
- Messages may be sent synchronously or asynchronously. An asynchronous send operation will return to the rest of the program as soon as the message has been queued to be sent; a synchronous send operation will only return once the message has actually been received by a corresponding receive operation.
- Messages may be sent and received as blocking or non-blocking operations. A blocking send/receive operation will only return to the rest of the program once the message has been sent/received; a non-blocking send/receive operation will return to the rest of the program immediately, allowing it to perform other tasks while checking back occasionally to see whether the message has been sent/received.

- As well as operations to send a single message between two processes in a communicator, collective operations involving all processes in a communicator are defined for several useful operations. For example, a broadcast sends data from a root process to all other processes in the communicator; a gather performs the inverse, sending data from all processes to a single root process; and a reduce performs a mathematical operation (e.g. sum, product, minimum, maximum, logical AND, etc.) across data from all processes, returning the result to a single process.

4.1.4 LAM/MPI

LAM/MPI (Local Area Multicomputer) [55] is a free open-source implementation of the MPI standard. It includes libraries which implement the MPI subroutines, as well as numerous utilities for starting up and shutting down a LAM/MPI environment, and for executing, monitoring and debugging MPI programs. Its modular System Services Interface (SSI) allows it to be configured to a multitude of different cluster architectures.

4.1.5 FFTW

FFTW (the Fastest Fourier Transform in the West) [48] is a library of algorithms for efficiently computing the discrete Fourier transform. It implements many different FFT (fast Fourier transform) algorithms to reduce the number of arithmetic operations. FFTW is unique, however, in that it adaptively selects the appropriate combination of algorithms to give optimum performance for the particular computer processor on which it is running. This feature makes its performance very portable between different computers and computer architectures.

4.1.6 C

C [40] is *the* ubiquitous general-purpose programming language. It provides basic data types to represent characters, integer and floating point numbers, pointers which represent the memory addresses of other data, structures which group different data types together as a single unit, and arrays which are multiple instances of a single data type. It provides arithmetical, logical, relational and other operators with which to manipulate these data types. It provides constructions which allow the evaluation of an expression to decide whether to execute or re-execute a block of commands, thus enabling a program to modify its own execution sequence. And it allows blocks of commands to be encapsulated as functions, which can then be called, receive data from and return data to other parts of the program. Beyond these basic elements, C provides no direct support for more sophisticated functionality, such as manipulation of an entire array or string of characters, memory allocation and management, and file input/output. Such functionality must either be handled directly by the programmer, or else is provided by explicitly-called functions, such as those provided by the C standard libraries. Nevertheless the limited number of basic elements have had the advantage of making C easy to learn, extremely portable and extensively optimisable.

4.1.7 The GNU Compiler Collection

The GNU Compiler Collection [56] is a free open-source collection of compilers for a number of programming languages, including C (the compiler for which is named `gcc`). It

is part of the GNU Project, which aims to build a complete open-source operating system using free open-source software, and which is closely associated with the Linux operating system.

4.2 A parallel data processing program

This section presents a parallel data processing program, written by the author in C using the MPI interface. It was compiled with LAM/MPI version 7.0.5, FFTW version 2.1.5, and the FrameL library version 6.14, using the gcc version 3.4.0 compiler. It was tested and finally executed on the ACIGA Data Analysis Cluster (ADAC) [57], a local cluster of 8 PCs attached via a high-speed network to 3 dual-processor servers.

A principal concern to the author was to create a functioning program within a short period of time, in order to devote as much of the remaining time as possible to processing the data. It is freely acknowledged that the design of the program presented below is not the most suitable, nor is its construction the most elegant. It is hoped that the accompanying criticism of its features and flaws will demonstrate the benefit of hindsight and experience.

4.2.1 The header

Most C programs include a header section. The header contains directives to include the headers for required libraries, definitions of new data types and data structures, and declarations of any global variables. The header directs the compiler to include the headers for a number of standard C libraries, as well as for the libraries of LAM/MPI, FFTW, FrameL and Frv. The Frv (Frame vector) library [58] is an addition to the FrameL library which provides additional functions to manipulate the vectors of data stored in Frame files. The header is included as listing A.1 in section A.1.1.

The following sections detail the main subroutines of the program. Other subroutines and utility functions are included as listings in section A.1.7.

4.2.2 The main subroutine

All C programs start by executing the main subroutine

Listing 4.1: The main subroutine

```
| int main(int argc, char *argv[ ]) {
```

(See listing A.2 in section A.1.2 for variable declarations). Its arguments contain any command-line arguments passed to the program when it was executed. C variable declarations usually immediately proceed the start of a subroutine or function. The first executable statement of a program using the MPI interface must call the subroutine

Listing 4.2: The main subroutine (continued)

```
| MPI_Init(&argc, &argv);
```

which properly initialises the MPI environment. The programs then calls

Listing 4.3: The main subroutine (continued)

```
| MPI_Comm_rank(MPI_COMM_WORLD, &rank);
| MPI_Comm_size(MPI_COMM_WORLD, &size);
| MPI_Comm_group(MPI_COMM_WORLD, &group);
```

to determine its rank within the default communicator `MPI_COMM_WORLD`, the size of the communicator (i.e. how many copies of the program are running) and a group object representing the processes within the communicator.

The program requires a number of command-line arguments. The first argument should specify the name of the channels file, a text file containing basic information about the channels of data to be analysed; the remaining arguments should specify the names of text files containing lists of names of Frame files to be analysed from each physical environment monitoring station. For example, the channels file for this project contained

```
256 SEISX SEISY SEBDCE01 SEBDCE02
256 SEISZ SEBDCE03
2048 LINE V1 power50
2048 MAG MABDCE
```

The first token of each line gives the sampling rates for the channels; the remaining tokens on each line give a distinctive part of the names of the Frame `FrAdcData` structures which should be searched for. In the above example, the program is instructed to analyse four channels. The first channel, sampled at 256 Hz, will be the average of the seismometer longitudinal (X) and latitudinal (Y) directions from a particular physical environment monitoring station: ANU, LIGO Hanford, LIGO Livingston, or VIRGO. The second channel, also sampled at 256 Hz, will be the seismometer vertical (Z) direction. The third and fourth channels, both sampled at 2048 Hz, will be the mains voltage monitor signal, and the average of all the magnetic field sensor directions. It was decided to average together some of the directions in order to reduce the computational load down from a possible 7 channels; it was also recognised that the actual seismometers and magnetic field sensors scattered around the world will undoubtedly not be aligned in the same directions. In the case of the seismometers, the division into horizontal (X, Y) and vertical (Z) corresponds to actual classes of seismic vibrations [59].

The program requires a fixed number of processes p to operate, the number being a function of the number of physical environment monitoring stations s it is being asked to analyse:

$$p = s(s + 1). \quad (4.1)$$

The program computes how many processes it needs and will throw an error if this number does not match the actual number of processes present:

Listing 4.4: The main subroutine (continued)

```
if (argc > 2) {
    COLLECTORS = argc-2;
    PROCESSORS = COLLECTORS;
    CROSSPROCESSORS = COLLECTORS*(COLLECTORS-1)/2;
    SIZE = COLLECTORS + PROCESSORS + 2*CROSSPROCESSORS;
}
else {
    return error(rank, 0, "At least one channel list and one file list must
        be supplied");
}
if (size != SIZE) {
    return error(rank, 0, "For %i file lists, %i nodes are required",
        COLLECTORS, SIZE);
```

```
| }
```

The processes are each designated a specific task, depending on their rank within `MPI_COMM_WORLD`. “Collectors” are responsible for reading in Frame files from one physical environment monitoring station each and sending the data to “processors” and “cross-processors”. Processors process data from only one station; two¹ sets of cross-processors perform correlated analyses between stations. Processes with the smallest ranks are designated collectors, then processors, then cross-processors as required.

The program creates a number of new communicators. One set of communicators are created to allow the collectors and the processors/cross-processors to communicate with each other separately:

Listing 4.5: The main subroutine (continued)

```
| MPI_Comm_split(MPI_COMM_WORLD, rank < COLLECTORS, 0, &local_comm);
```

This subroutine will return one of two new communicators in the variable `local_comm`, depending upon whether the expression `rank < COLLECTORS` is true (for the collectors) or false (for the processors/cross-processors).

Another set of communicators are created for each collector to communicate with and send data to the appropriate processors and cross-processors. They are constructed such that, as per their definitions, each processors will communicate uniquely with only one collector, and each cross-processor will communicate with a unique combination of two collectors. An example for four physical environment monitoring stations is shown in table 4.1. The permutations of collectors assigned to each cross-processors is found by reading down the columns; the particular cross-processors contained in the communicator assigned to each collector is found by reading across rows. The code is included as listing A.3 in section A.1.2.

This structure of a fixed number of processes, each with a designated task, was originated motivated by its simplicity; once the communicators between collectors and processors/cross-processors are set up, the overall structure of the program is fixed and need no longer be known by the remainder of the program. Despite this, it is in retrospect a serious design flaw, for a number of reasons. One reason pertains to load balancing, an essential consideration for parallel programs which states that work should be distributed as evenly as possible across all processes, or at least across all processes that work synchronously. Unfortunately this turned out not to be the case for the implemented analyses; the correlated analyses were split over two sets of cross-processors in a belated attempt to rectify this, but regardless a large load imbalance remained, causing a significant performance cost. The requirement for a fixed number of nodes in squared proportion to

¹The reason for this is explained below

(New) Ranks	Processor				Cross-processors					
	4	5	6	7	8/14	9/15	10/16	11/17	12/18	13/19
0 (0)	(1)				(2/5)	(3/6)	(4/7)			
1 (0)		(1)			(2/5)			(3/6)	(4/7)	
2 (0)			(1)			(2/5)		(3/6)		(4/7)
3 (0)				(1)			(2/5)		(3/6)	(4/7)

Figure 4.1: Example for four physical environment monitoring stations of the structure of the collector-to-processor/cross-processor communicators. Numerals in brackets refer to ranks in the new communicators; those without refer to ranks in `MPI_COMM_WORLD`.

the number of physical environment monitoring stations being analysed meant that, for 4 stations, the 20 processes required exceeded ADAC's 14 computer processors. Scheduling multiple processes on some computer processors was therefore required, which was cumbersome and affected performance.

After processing the channels file (see listing A.4 in section A.1.2), the program creates a touch file:

Listing 4.6: The main subroutine (continued)

```

| if (rank == 0) {
|     fclose(fopen(TOUCH_FILENAME, "wb"));
| }

```

This facilitates a very simple way of communicating a user command to the program, which will be running in the background and therefore not possess any user interface. In this case, deleting the touch file will cause the program to terminate cleanly at the next possible opportunity.

The program calls several subroutines to initialise the long and short timescale analyses, and then finally transfers control to one of two subroutines which implement the collectors and the processors/cross-processors (see listing A.5 in section A.1.2). Once these subroutines return, it cleans up any dynamically-allocated resources (a usual house-keeping routine for C programs), calls the `MPI_Finalize` subroutine to properly finalise the MPI environment, and exits (see listing A.6 in section A.1.2).

4.2.3 The collector subroutine

The collector subroutine takes as arguments the communicator of all collectors; the communicator of the processor and cross-processors to which it will be sending data; the number of channels, and an array containing the sampling rates for each channel; the size of an array of the structure type `StringIndex`, containing the names of the Frame `FrAdc-Data` structure which should be searched for and the channels they correspond to, and the array itself; and the name of the text file containing the list of names of the Frame files to be analysed from one of the physical environment monitoring stations:

Listing 4.7: The collector subroutine

```

| int collector(MPI_Comm local_comm, MPI_Comm to_proc_comm, int CHANNELS,
|             double sampling_rate[ ], int CHANNEL_NAMES, StringIndex *channel_names,
|             char *list_filename) {

```

(See listing A.7 in section A.1.3 for variable declarations). To initialise itself the subroutine first gets its rank within and the size of the communicator of all collectors:

Listing 4.8: The collector subroutine (continued)

```

|     MPI_Comm_rank(local_comm, &local_rank);
|     MPI_Comm_size(local_comm, &local_size);

```

The subroutine then broadcasts the name of the list file to the processors and cross-processors, which will use it to identify the files they create to store the results of their analyses:

Listing 4.9: The collector subroutine (continued)

```

|     for (i = strlen(list_filename); (i > 0) && (isalnum(list_filename[i-1])); i--);

```

```
strcpy(line, &list_filename[i]);
MPI_Bcast(line, LINE, MPI_CHAR, 0, to_proc_comm);
```

After initialising the buffers that will be used to store the data from the Frame files, and creating an output list file to store the names of the Frame files that have been processed (see A.8 in section A.1.3), the subroutine opens the input list file and begins to read the name of each Frame file in turn, line by line, using the C library function `fgets`. It also tests the value of the variable `coll_continue` which determines whether the collector should continue processing; if it is set to 0 (equivalent to false), the C logical AND operator `&` will terminate the loop:

Listing 4.10: The collector subroutine (continued)

```
flist = fopen(list_filename, "r");
coll_continue = 1;
while ((fgets(line, LINE, flist) != NULL) & coll_continue) {
    line[strlen(line)-1] = '\0';
    while (access(line, F_OK) != 0) {
        sleep(1);
    }
};
```

The UNIX `access` function determines whether or not the file exists; the subroutine will halt until this function returns true. This functionality was originally included for the purpose of performing data processing concurrent to data acquisition, but was never actually needed.

The subroutine now opens the Frame file and begins to cycle through all the frames, represented by `FrameH` structures, in the file. The **while** loop also tests the value of the `coll_continue` variable. At each iteration it uses the UNIX `access` function to determine whether the touch file still exists; if it does not, it sets the `coll_continue` variable to 0. The **continue** statement causes the loop to immediately proceed to its next iteration, where the `& coll_continue` condition will fail and the loop will terminate:

Listing 4.11: The collector subroutine (continued)

```
frfile = FrFileNew(line);
frame = NULL;
while (((frame = FrameReadRecycle(frfile, frame)) != NULL) & coll_continue) {
    if (access(TOUCH_FILENAME, F_OK) != 0) {
        coll_continue = 0;
        continue;
    }
}
```

The subroutine will now iterate through any `FrAdcData` structures in the frame, searching the name of each structure for one of the names in the channel names array. If a match is found, the data is copied from the frame file to the appropriate buffer (see listing A.1.3 in section A.1.3). The `FrvZeroMean` Frv library subroutine reduces the data to zero mean, as this was found by prior experimentation in MATLAB to give better results than otherwise for correlation studies by ameliorating the inconsistent scaling of data from the four physical environment monitoring stations.

C does not provide class structures to encapsulate data and subroutines associated with it into a single unit, therefore the buffers were implemented as a `Buffer` structure type (see listing A.1 in section A.1.1) together with several `buffer...` subroutines that manipulated the structure (and which are listed in section A.1.7). In general this is a very poor

programming paradigm, as the `Buffer` structure containing the data could accidentally be manipulated in a way that might invalidate it. Encapsulation can prevent this situation from arising, by making the data accessible only through the subroutines of a class, in which the data is contained. The C++ [40] programming language provides a very powerful class structure among many other features for more structured, object-oriented programming (OOP). Although at the time it was felt that the increased design time required for a well-implemented C++ program would be a hindrance to rapid implementation, and other reasons such as the slightly simpler nature of the MPI interface under C than under C++ resulted in the author adopting C, a more considered re-implementation would undoubtedly be written in C++.

After iterating through all the `FrAdcData` structures in each frame, the subroutine checks to see if all buffers contain at least some data:

Listing 4.12: The collector subroutine (continued)

```
all_have_data = 1;
for (i = 0; i < CHANNELS; i++) {
    all_have_data = all_have_data & (series[i].length > 0);
}
if (all_have_data) {
```

If this is true, it synchronises the starting GPS times of the channels across all the collectors, if it has not done so already, using an MPI all-reduce operation. This convenient feature means that the sets of Frame files being processed by the collectors need not themselves be synchronised to begin at the same GPS time, which was inevitably the case (see listing ?? in section A.1.3).

Once the buffers have been synchronised, the subroutine repeatedly checks to see if all buffers contain at least `SEND_SEGMENT` seconds of data:

Listing 4.13: The collector subroutine (continued)

```
coll_continue = 1;
while (all_have_data & coll_continue) {
    all_have_data = 1;
    for (i = 0; i < CHANNELS; i++) {
        all_have_data = all_have_data & (series[i].length >= SEND_SEGMENT);
    }
}
```

If this is true, then the collector is deemed to have sufficient data to send off to the processors and cross-processors. At this point the subroutine calls an MPI all-reduce operation over all of the collectors' `coll_status` variables, combining them with the logical AND operator `MPI_LAND` and returning the result to all collectors as `coll_continue`. The `MPI_Allreduce` subroutine will wait for all collectors to reach this point before returning:

Listing 4.14: The collector subroutine (continued)

```
if (all_have_data & coll_continue) {
    coll_status = 1; coll_continue = 0;
    MPI_Allreduce(&coll_status, &coll_continue, 1, MPI_INT, MPI_LAND,
        local_comm);
```

If another collector has also reached this point, its `coll_status` variable will be 1. If it has read all of its Frame files, or if it detected the touch file had been deleted, its `coll_status`

variable will be 0, and thus the `coll_continue` variables of all the collectors will also be 0. If all collectors agree to continue, each collector will broadcast the channel number, the starting GPS time of the data, and the data itself to their respective processors and cross-processors (see listing A.10 in section A.1.3).

Once a collector has finished read all of its Frame files, it closes all open files, indicates to all other collectors and its respective processors/cross-processors that it has finished, cleans up any dynamically-allocated resources and returns to the `main` subroutine (see listing A.11 in section A.1.3).

4.2.4 The processor subroutine

The processor subroutine takes as arguments the size of an array containing the communicators of which the processor/cross-processor is a member (which will be 1 for a processor and 2 for a cross-processor) and the array itself; an integer indicating for a cross-processor which set it belongs to (which will be either 1 or 2); the number of channels, and an array containing the sampling rates for each channel:

Listing 4.15: The processor subroutine

```
int processor(int FROM_COLL_COMM, MPI_Comm from_coll_comm[ ], int crossproc_set,
             int CHANNELS, double sampling_rate[ ]) {
```

It first receives the file name(s) broadcast by the collector(s) with which it communicates, concatenating them into a single string which will be used to identify the files created by the analyses. It then initialises the buffers that will store data from the collectors (see listing A.13 in section A.1.4).

The subroutine then sets the `from_coll_comm_left` variable to the number of communicators, and hence the number of collectors communicating with the processor. This variable keeps track of how many collectors are still active. Whenever a collector indicates that it has finished reading Frame files, the `from_coll_comm_left` variable is reduced by 1. Once the count is zero, the processor can safely terminate:

Listing 4.16: The processor subroutine (continued)

```
from_coll_comm_left = FROM_COLL_COMM;
while (from_coll_comm_left > 0) {
```

The subroutine now iteratively checks each communicator to see whether there is data to be received from the corresponding collector. The `from_coll_comm_stopped` array keeps track of specifically which collectors are still active:

Listing 4.17: The processor subroutine (continued)

```
for (j = 0; j < FROM_COLL_COMM; j++) {
    if (!from_coll_comm_stopped[j]) {
        i = 0;
        while (i > PROC_CONTINUE) {
            MPI_Bcast(&i, 1, MPI_INT, 0, from_coll_comm[j]);
```

As long as the variable `i` receives from the collector a value larger than the constant `PROC_CONTINUE`, it represents a valid channel number: the processor will then receive each channel in turn and store the data it in the appropriate buffers (see listing A.14 in section A.1.4).

Once the variable `i` receives `PROC_CONTINUE`, the loop is broken and the processor will move on to checking the next collector. If the variable `i` receives `PROC_STOP`, the collector is removed from the check-list:

Listing 4.18: The processor subroutine (continued)

```

    }
    if (i == PROC_STOP) {
        from_coll_comm_stopped[j] = 1;
        from_coll_comm_left--;
    }
}
}
}

```

If all collectors are still active, then the processor or cross-processor may proceed with the execution of the analyses (see listing A.15 in section A.1.4). Once all collectors are removed from the check-list, the processor/cross-processor cleans up any dynamically-allocated resources and returns to the main subroutine (see listing A.11 in section A.1.3).

4.2.5 Implementation of the long timescale analyses

This subsection presents the implementation of the long timescale analyses described in section 3.1.

The discrete autocorrelation (equation 3.2) is implemented by the `doLongAuto` subroutine, which takes as arguments the `Buffer` which stores the data, the sampling rate of the channel, the number of the channel, and a string used to identify any data files. The subroutine first attempts to initialise its variables from a data file, if the file exists, and then calculates the autocorrelation of the data for $0 \leq \tau \leq \text{LONG_CORR_SHIFT}$ seconds, adding it element-wise to any previously calculated autocorrelation. It then writes its data to file and returns. It is included as listing A.17 in section A.1.5.

The cross spectrum (equation 3.28) of discrete data is implemented by the `doLongCross` subroutine in a manner analogous to the `doLongPower`. It is included as listing A.18 in section A.1.5.

The discrete power spectrum (equation 3.27) is implemented by the `doLongPower` subroutine, which takes similar arguments to the previous subroutines. The subroutine first attempts to initialise its variables from a data file, if the file exists. Then, as long as series is at least `LONG_SPECT_WINDOW * LONG_SPECT_OVERLAP` seconds in length, it calculates the power spectrum as follows.

The subroutine first loads the data into a pre-allocated buffer and multiplies the data by a pre-calculated Bartlett window, for the purposes of data windowing as discussed on page 26. The pre-allocation and pre-calculation are contained in the `longInit` subroutine (see listing A.34 of section A.1.7). The Fourier transform is then computed by `FFTW` using a pre-compiled plan encoding the optimal combination of FFT algorithms for the computer processor on which `FFTW` is executing. The Fourier transform returned by the `rfftw` subroutine is performance reasons a complicated combination of real and imaginary components [48], which are deciphered in the computation of the power spectrum by the `algrCross` subroutine (see listing A.24 in section A.1.7), which is added element-wise to any previously calculated power spectrum. The calculation is then repeated if the series is still of sufficient length. The data is shifted by only `LONG_SPECT_WINDOW` to create an overlap of `LONG_SPECT_OVERLAP`, as discussed on page 27. Once there is no longer

sufficient data to compute the power spectrum, the subroutine writes its data to file and returns. It is include as listing A.19 in section A.1.5.

The cross spectrum (equation 3.28) of discrete data is implemented by the `doLongCross` subroutine in a manner analogous to the `doLongPower` subroutine. It is included as listing A.20 in section A.1.5.

4.2.6 Implementation of the short timescale analyses

This subsection presents the implementation of the short timescale analyses described in section 3.2.

The discrete power spectrogram and cross spectrogram (equations 3.42 and 3.41) were implemented by the `doTrnsPower` and `doTrnsCross` subroutines respectively.

The `doTrnsPower` subroutine takes as arguments the `Buffer` which stores the data, the sampling rate of the channel, the `Buffer` in which the spectrogram is stored, the spectrogram “sampling rate” (as in the number of spectrogram elements per unit time), and the number of the channel. The computation of the power spectrum is analogous to the `doLongPower` subroutine. It is included as listing A.21 in section A.1.6.

The `doTrnsCross` subroutine is analogous to the `doTrnsPower` and `doLongCross` subroutines; it is included as listing A.22 in section A.1.6.

The search algorithm described in section 3.2.3 was implemented by the `doTrnsSearch` subroutine. It takes as arguments the `Buffer` in which the spectrogram is stored and its associated “sampling rate”, the sampling rate of the channel, the number of the channel, and a string used to identify data files: It firstly attempts to read in a number of data files, if they exist. Then, while the spectrogram remains long enough, it implements the search algorithm as follows.

The subroutine copies the spectrogram into a number of arrays corresponding to the current “event” bin $\mathbf{B}_{f,t}$ and the comparison “search” bins $\mathbf{B}_{f,t-dt_{\max}}, \dots, \mathbf{B}_{f,t-1}, \mathbf{B}_{f,t+1}, \dots, \mathbf{B}_{f,t+dt_{\max}}$, where dt_{\max} is equivalent to `SRCH_TIME_RANGE`. The constants `SRCH_TIME_OVERLAP` and `SRCH_FREQ_OVERLAP` specify the overlapping of the bins, which will have dimensions of `SRCH_TIME_BIN` \times `SRCH_TIME_OVERLAP` spectrogram rows in time and `SRCH_FREQ_BIN` \times `SRCH_FREQ_OVERLAP` Hz in frequency.

The subroutine ?? (see listing A.39 in listing A.1.7) then computes the five test statistics listed in section 3.2 (equations 3.47, 3.48, 3.51, 3.53 and 3.55) and returns then in the `event_stat` array. For each test statistic, only the largest `STAT_SAVE` values are saved to the array `stat_list`. If any of the computed statistics is large enough, it is added to the end of `stat_save`, and then moved up through the array until it reaches its sorted position. If any of the computed statistics is large enough to be within the top `STAT_SPCTGM_SAVE` values, its associated spectrogram is saved as well. The subroutine continues to shift the spectrogram until the remainder is insufficient for further searches. It then writes all its data to files, and returns. It is included as section A.23 in section A.1.6.

Results

In the course of this project, some 183 gigabytes of physical environment monitoring data were acquired from four stations located around the world; at The Australian National University (ANU), at the LIGO Hanford observatory (LHO), at the LiGO Livingston observatory (LLO), and at the VIRGO detector (VIRGO). A total of 38 days and 21 hours of data, originating from February to March 2003 and from August to October 2004, was processed in 35 days 4 hours and 13 minutes. The processing was broken into four data sets, and is summarised in figure 5.1. The results were then post-processed using MATLAB (the code for which is included under section A.2), generating a total of 200 graphs and 1040 spectrograms.

This chapter presents a summary of the results of the data analyses, and attempts to identify any interesting features, particularly with regard to correlated signals.

5.1 Long timescale analyses

5.1.1 Power spectra

Before examining the results further it is useful to verify that the data from each station and on each channel has some consistent structure across the range of the data set; if not, then the originating instrument is likely to be non-functional. This can be done using the power spectra.

Figure 5.2 (page 47) shows the power spectra of the ANU seismometers. The horizontal seismometer shows a consistent structure, with some minor movement of peaks in the higher frequencies; the vertical seismometer, however, was clearly non-functional for figures

Data set	α	β	γ	δ
Stations involved: ANU	✓	✓		✓
LHO	✓		✓	✓
LLO	✓		✓	✓
VIRGO		✓	✓	✓
Start time: UTC	27/02/03 09:23:07	06/08/04 05:59:47	20/09/04 06:42:27	04/10/04 23:59:47
End time: UTC	14/03/03 05:29:47	16/08/04 04:34:47	29/09/04 01:12:51	10/10/04 07:48:07
Duration of data set	14d 20h 7m	9d 22h 35m	8d 18h 30m	5d 7h 48m
Computation time	7d 17h 20m	10d 4h 47m	10d 4h 47m	7d 1h 19m

Figure 5.1: Summary of the performed data processing.

5.2(d) and 5.2(f).

Figure 5.3 (page 48) shows the power spectra of the ANU mains voltage monitor and magnetic field sensor. The mains voltage monitor is nearly two orders of magnitude quieter in the lower two figures 5.2(c) and 5.2(e), while still showing prominent harmonics at 50 Hz, the Australian mains frequency. The magnetic field sensor is also quieter, but the shape of its spectrum has been morphed from flat to corrugated in the lower two figures, with more substantial noise at low frequencies.

Figure 5.4 (page 49) shows the power spectra of the LHO seismometers. The most recent spectra 5.4(e) and 5.4(f) have a much more substantial background noise compared with previous spectra averaged over a similar period. Otherwise many structures, such as the prominent line at 60 Hz, the United States mains frequency, are clearly distinguishable in all the figures.

Figure 5.5 (page 50) shows the power spectra of the LHO mains voltage monitor and magnetic field sensor. The noise in the mains voltage monitor is reduced substantially from the first figure 5.5(a) to the lower figures 5.5(c) and 5.5(e). The symmetric structure in 5.5(b) is a clear example of aliasing. The noise is reduced in 5.5(d), but reappears in 5.5(f). It is not clear whether this was caused by the instrument or by its environment.

Figure 5.6 (page 51) shows the power spectra of the LLO seismometers. These spectra are flat and featureless compared with the Hanford spectra, suggesting a significantly different local seismic noise environment. Otherwise there are no significant changes over the three data runs.

Figure 5.7 (page 52) shows the power spectra of the LLO mains voltage monitor and magnetic field sensor. The mains voltage monitor spectra change from flat to being slightly curved, then substantially curved (figures 5.7(a), 5.7(c) and 5.7(e)). Most importantly though, the 60 Hz harmonics disappear completely, indicating that something is seriously wrong. The magnetic field sensors show qualitatively similar distortions, except that the 60 Hz harmonics remain. This suggests that it is likely that the spectra are environmental in origin; nevertheless, the mains voltage monitor is likely to be non-functional.

Figure 5.8 (page 53) shows the power spectra of the VIRGO seismometers. These spectra show almost identical agreement across the three data sets, indicating a more stable seismic environment. Note the subtle differences in the horizontal and vertical seismometer spectra, for example, the large mound from 60–70 Hz is present only in the right-hand figures.

Figure 5.9 (page 54) shows the power spectra of the VIRGO mains voltage monitor and magnetic field sensor. These spectra also show close agreement across the three data sets. There is a slight reduction in noise from the first mains voltage spectrum (figure 5.9(a)) to figures 5.9(c) and figures 5.9(e), and a similar but much more substantial noise reduction in the magnetic field sensor spectra (figure 5.9(b) to figures 5.9(d) and 5.9(f)).

5.1.2 Autocorrelations

Some interesting features were found in the autocorrelation analyses. Figure 5.10(a) (page 55) shows an interesting “wave-packet” structure, indicating that the ANU horizontal seismometer is correlated with itself over short timescales of less than 0.5 seconds. This structure is also present, although with a greater background oscillation, in the autocorrelation of a later data set (figure 5.10(b)) and is only very slightly present in the vertical direction (figure 5.10(c)). Figure 5.10(d) of the non-functional vertical seismometer indicates a random signal, with the sharp peak corresponding to the instant when the signal

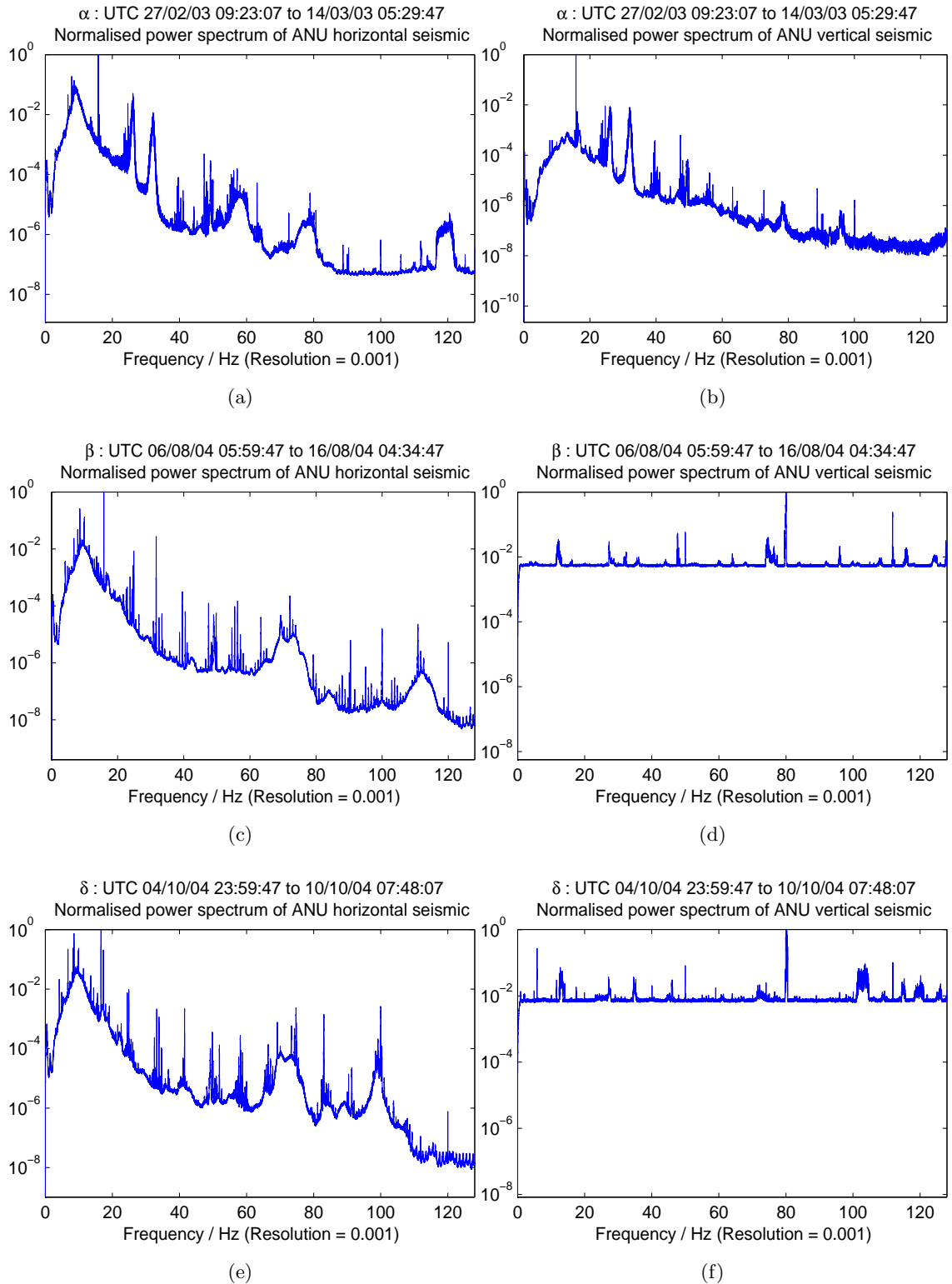


Figure 5.2: Power spectra of the ANU horizontal and vertical seismometers.

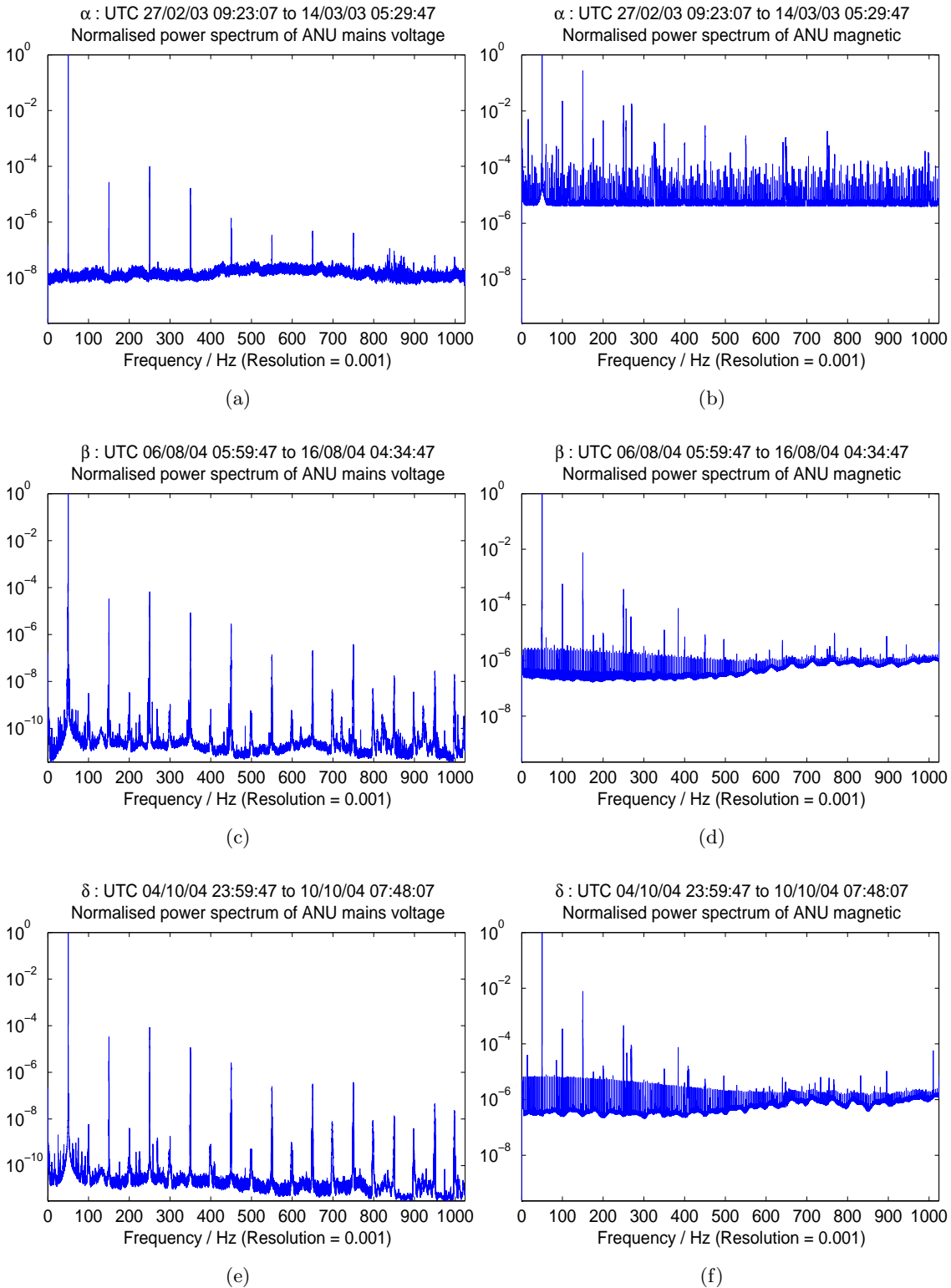


Figure 5.3: Power spectra of the ANU mains voltage monitor and magnetic field sensors.

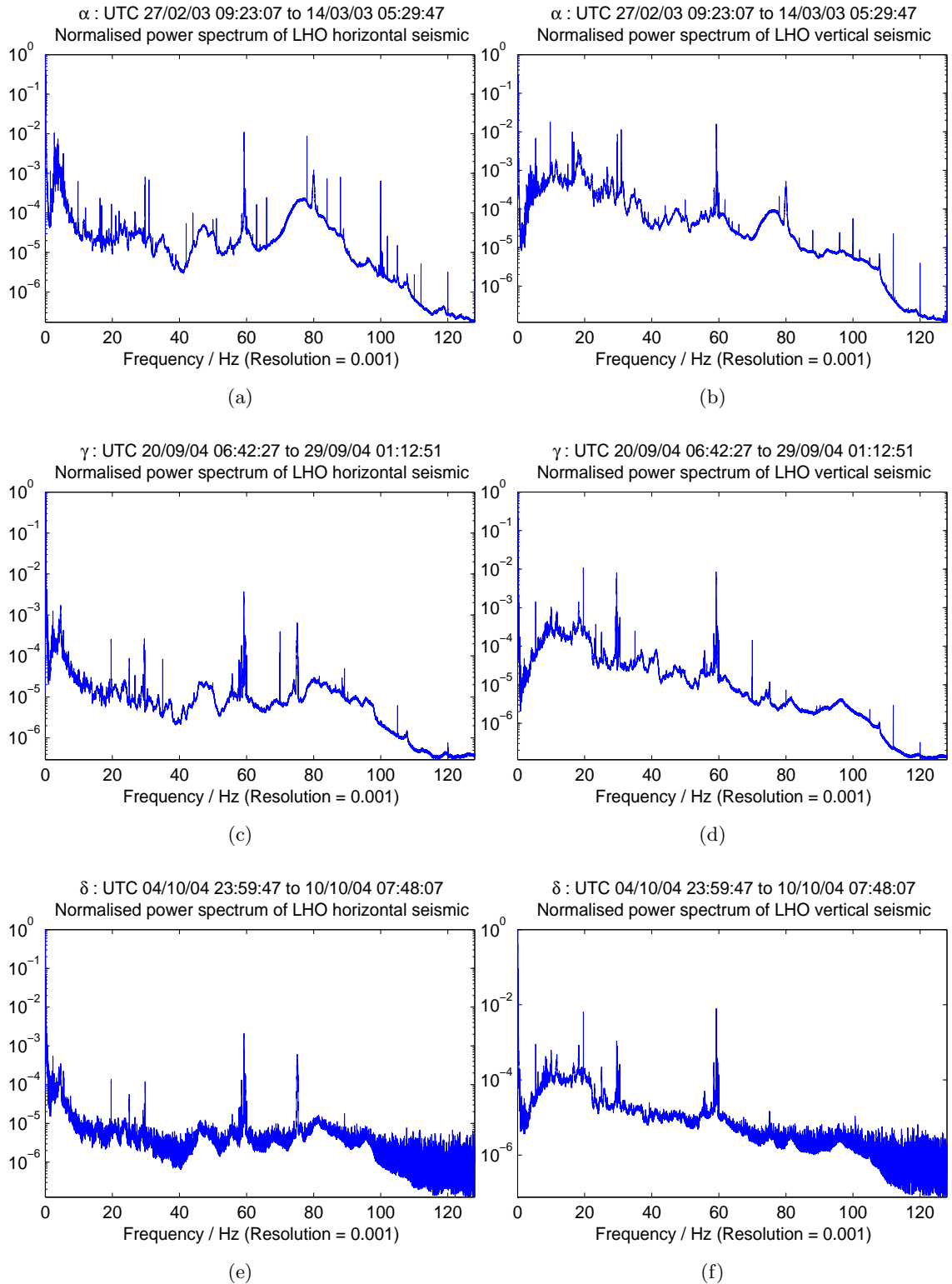


Figure 5.4: Power spectra of the LHO horizontal and vertical seismometers.

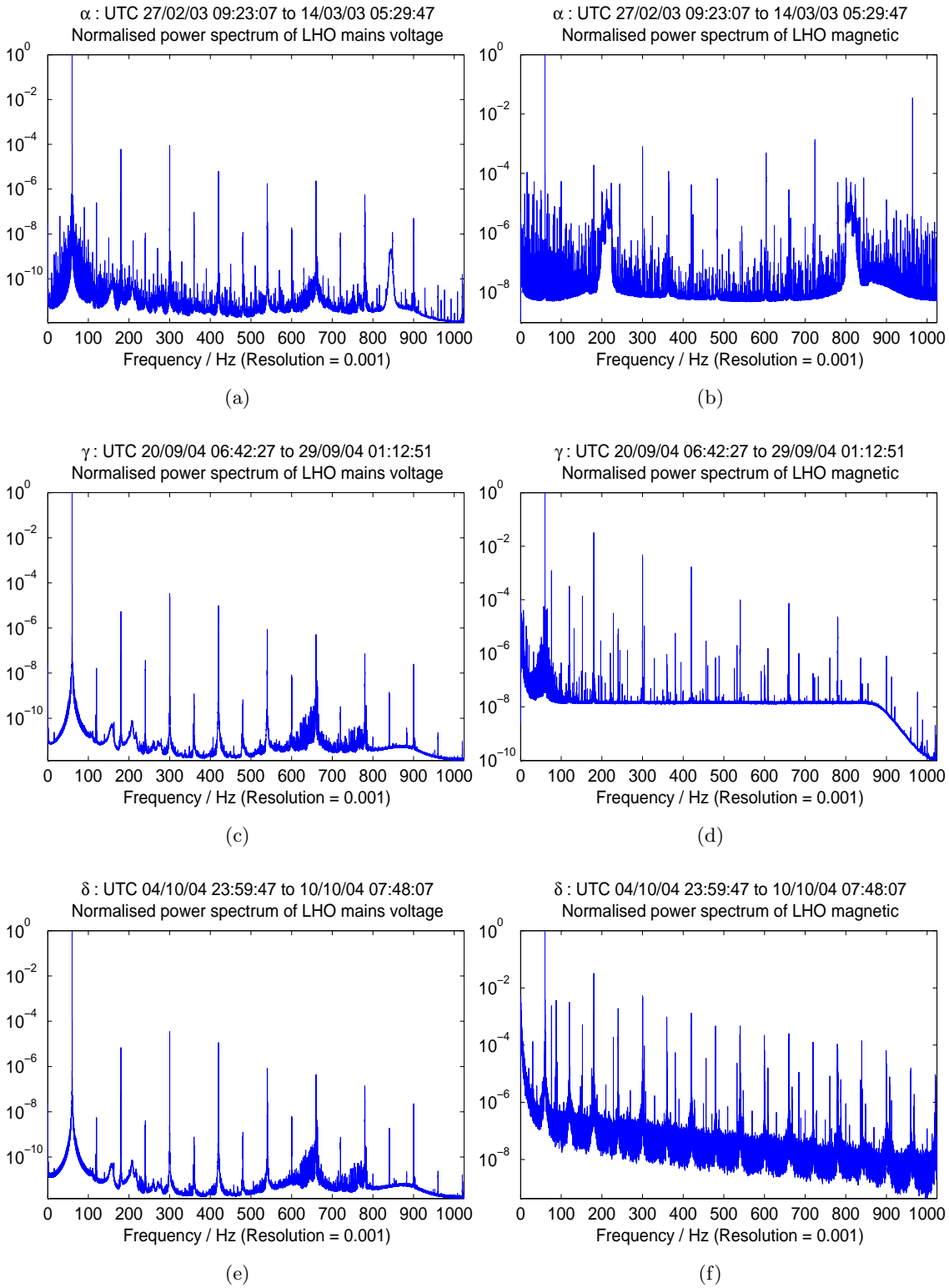


Figure 5.5: Power spectra of the LHO mains voltage monitor and magnetic field sensors.

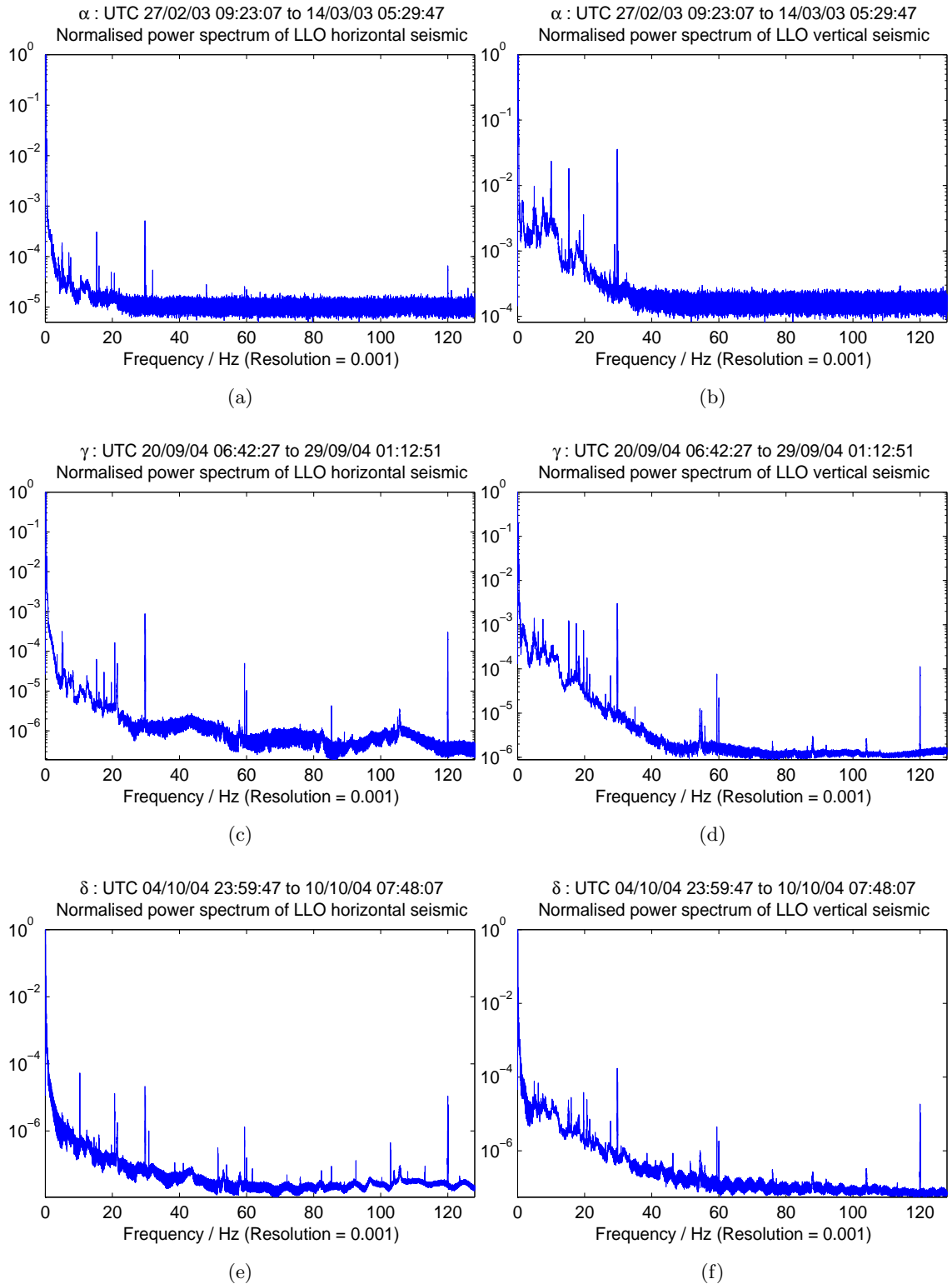


Figure 5.6: Power spectra of the LLO horizontal and vertical seismometers.

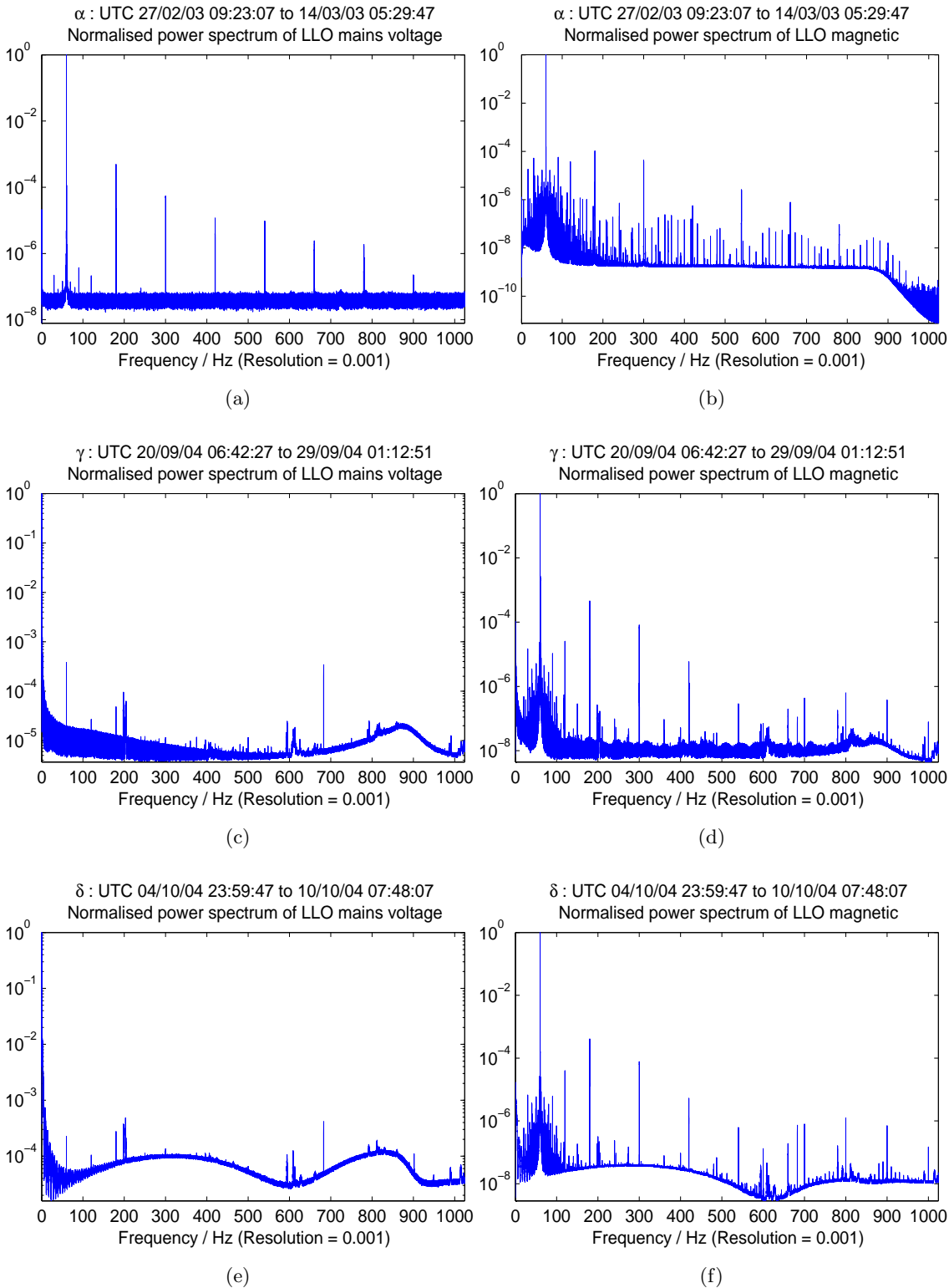


Figure 5.7: Power spectra of the LLO mains voltage monitor and magnetic field sensors.

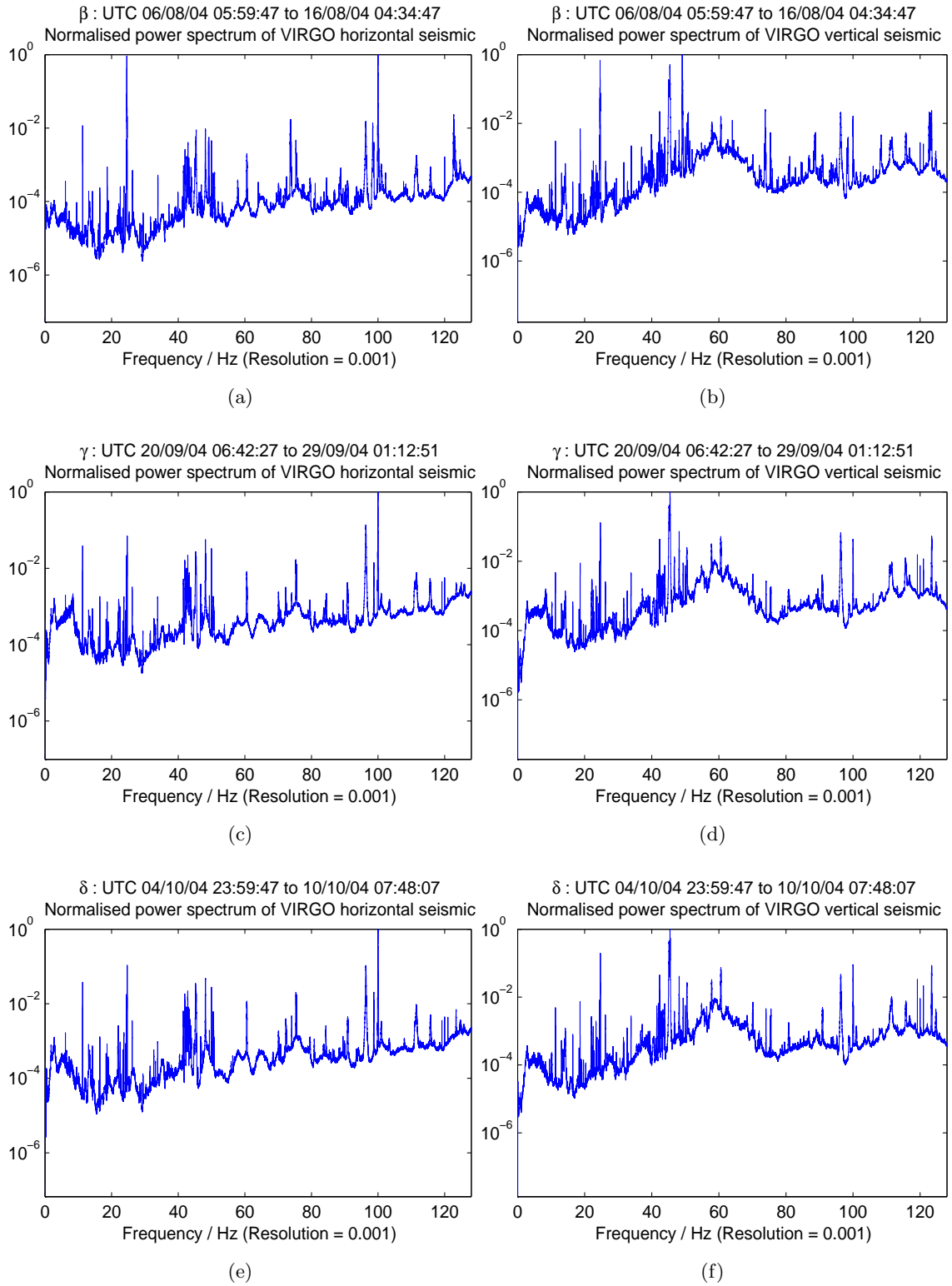


Figure 5.8: Power spectra of the VIRGO horizontal and vertical seismometers.

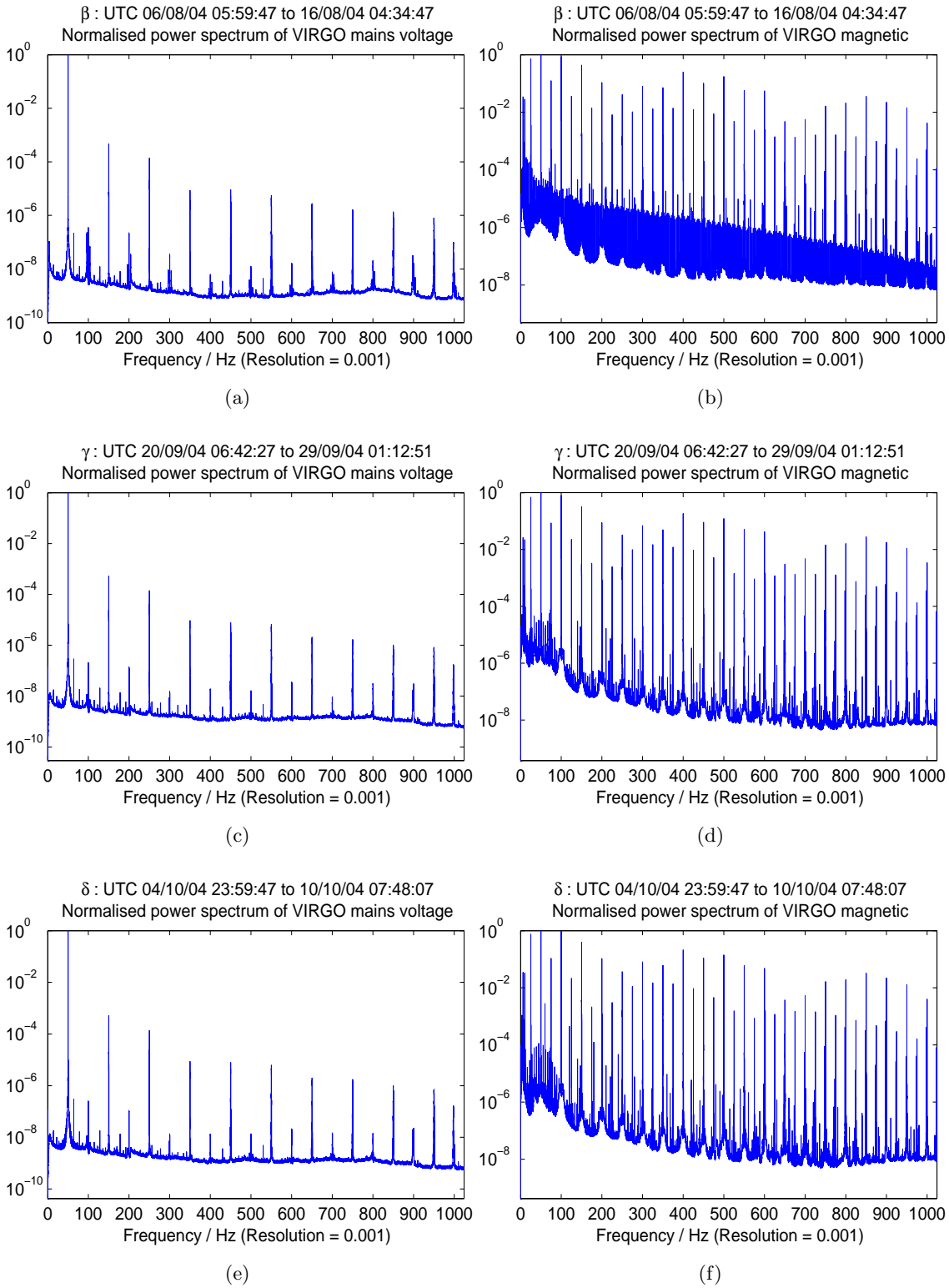


Figure 5.9: Power spectra of the VIRGO mains voltage monitor and magnetic field sensors.

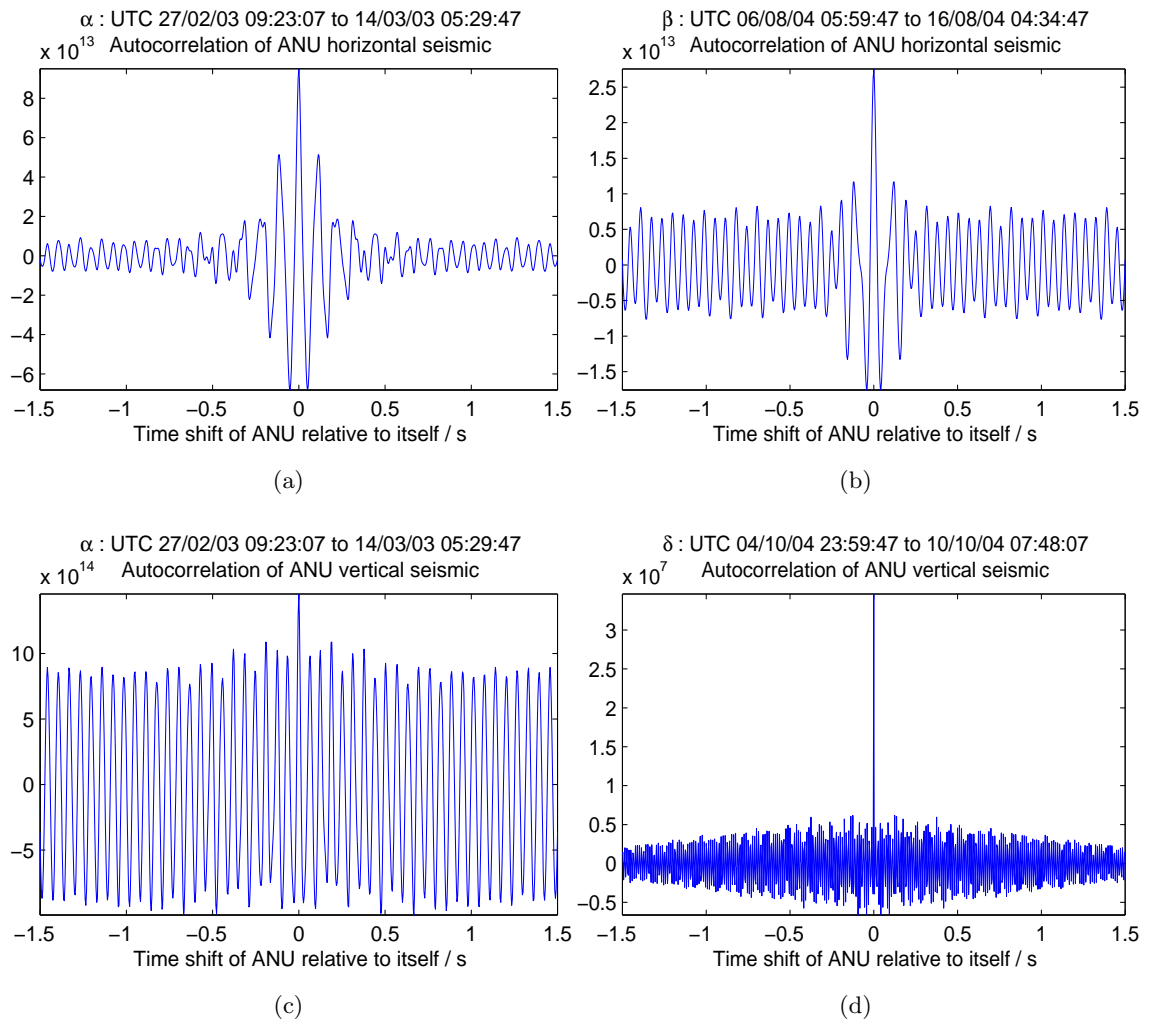


Figure 5.10: Autocorrelation of the ANU horizontal and vertical seismometers.

identically matches itself.

Figure 5.11(a) (page 56) shows irregular oscillations in the LHO horizontal seismometer autocorrelations, again suggesting some kind of short timescale self-coherence. The effect has, however, disappeared in the autocorrelations (figure 5.11(b)) of a later data set. Figure 5.12(a) and 5.12(b) show a similar but much smaller effect in the LLO vertical seismometers. The autocorrelation are also much smoother, likely due to a deliberate or accidental filtering process.

Figure 5.13(a) and 5.13(b) (page 56) would seem to indicate a fault in the LLO mains voltage monitor. While the first spectrum is dominated by the expected 60 Hz oscillations, the second spectrum shows a bizarre triangular shape with a sharp central spike. It is not clear what is implied of the signal by this shape of autocorrelation.

Figure 5.14 (page 57) shows autocorrelations of the VIRGO horizontal and vertical seismometers. It is interesting that the oscillating structure shown in figure 5.14(c) for the vertical seismometer is reproduced in the autocorrelation (figure 5.14(b)) of a later data set, except that this time it is for the horizontal seismometer. Figures 5.14(a) and 5.14(d) might also be considered in a similar way. One obvious answer is that the two channels may have somehow been switched, which could be easily verified; if not a more unusual

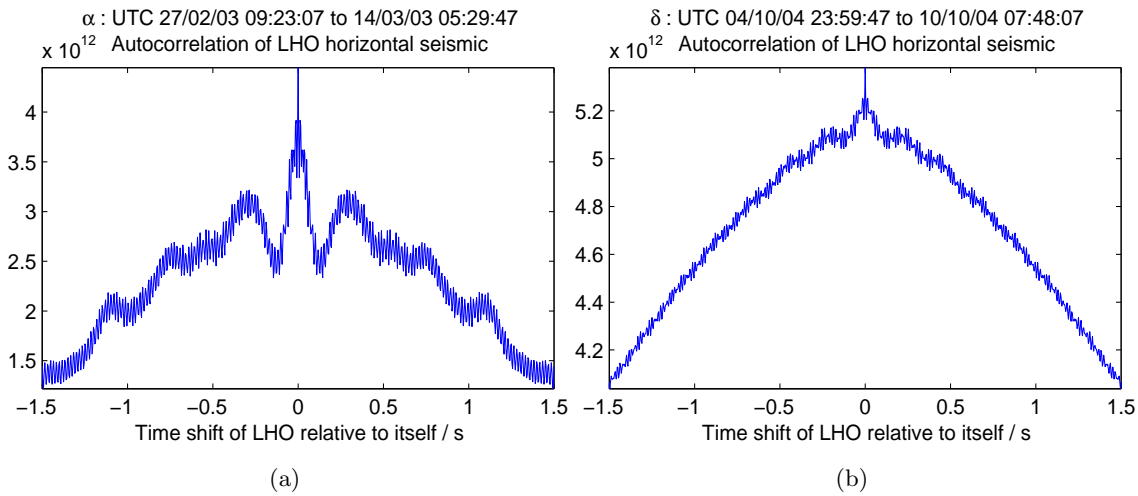


Figure 5.11: Autocorrelation of the LHO horizontal seismometer.

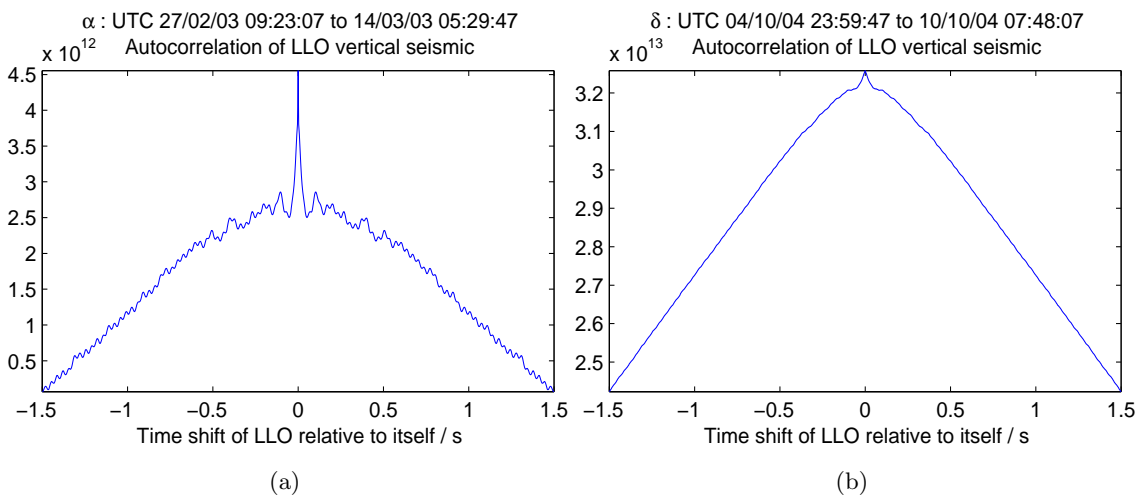


Figure 5.12: Autocorrelation of the LLO vertical seismometer.

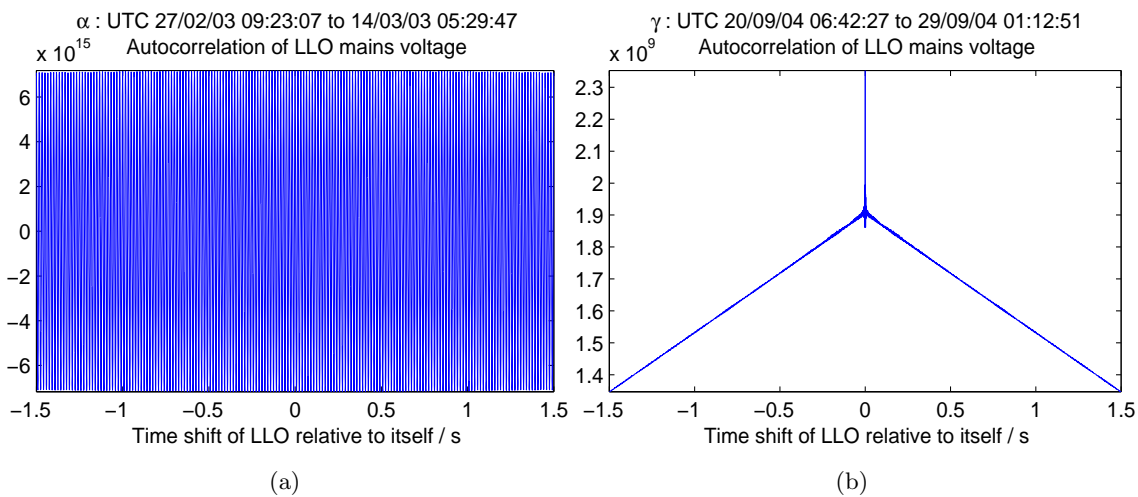


Figure 5.13: Autocorrelation of the LLO mains voltage monitor.

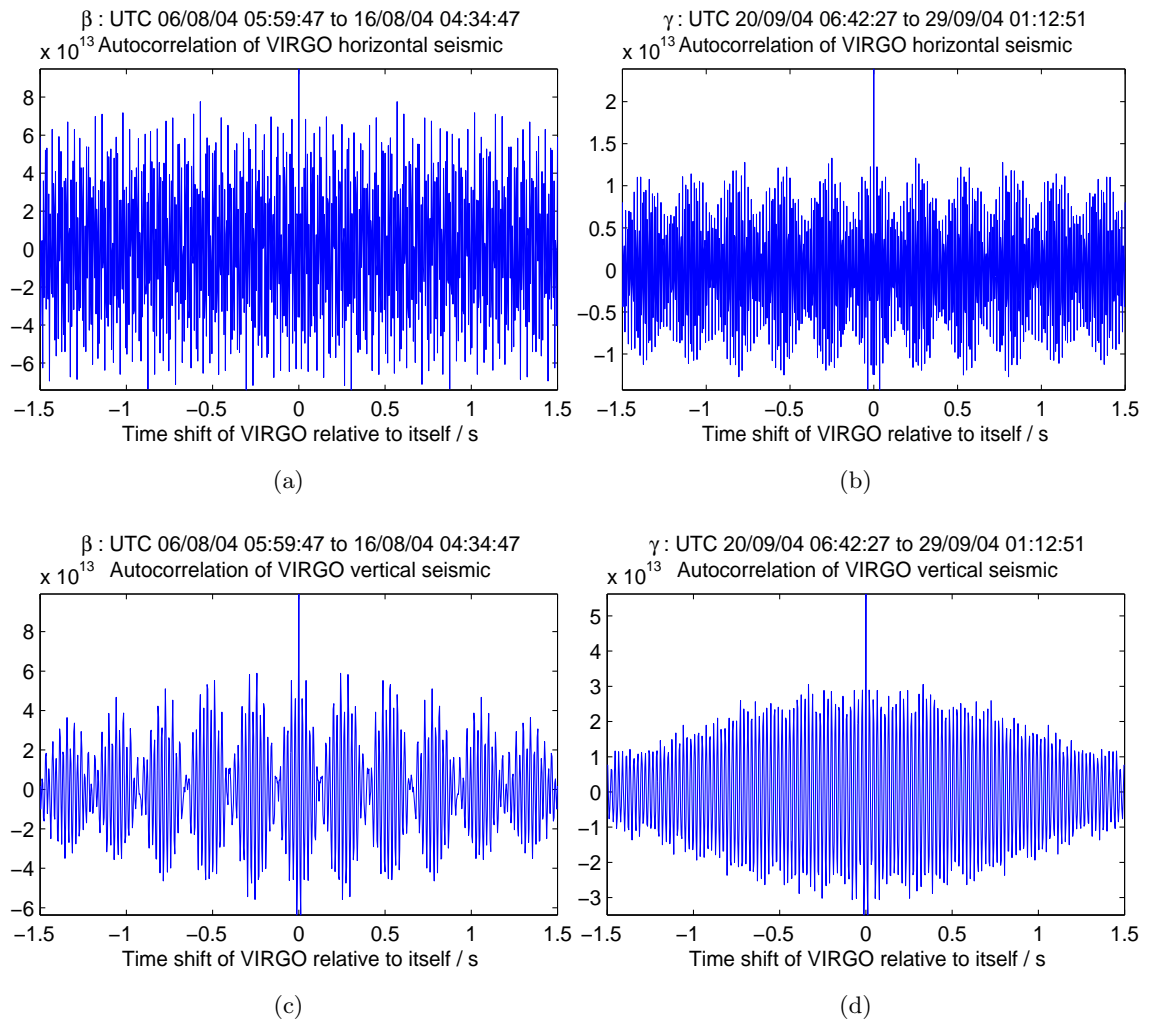


Figure 5.14: Autocorrelation of the VIRGO horizontal and vertical seismometers.

explanation would be required.

5.1.3 Correlations

Figure 5.15(a) and 5.15(b) (page 58) show a quantitative change in the oscillations of the ANU-LHO mains voltage monitors' correlations. As noted in chapter ??, asymmetries in oscillating correlations are indicative of two sinusoidal signals of differing frequencies. The change in the asymmetry of the correlation, as in the change in amplitudes of the oscillations, would seem to indicate a change in the frequencies or relative phases of the signals themselves.

Figure 5.16 (page 58) shows random correlations between the ANU-VIRGO horizontal and ANU-LLO vertical seismometers. Notice that unlike autocorrelations of random data there is no central peak where the signal exactly matches itself. Little information can be gained from these correlations, except to note that the ANU-VIRGO correlations appear to be significantly noisier than the ANU-LLO correlations.

Figure 5.17 (page 60) shows correlations between LHO and LLO horizontal and vertical seismometers. As was the case for figures 5.12(a) and 5.12(b) (page 56), they are surprisingly smooth, a deliberate or accidental filtering process likely to be responsible. It

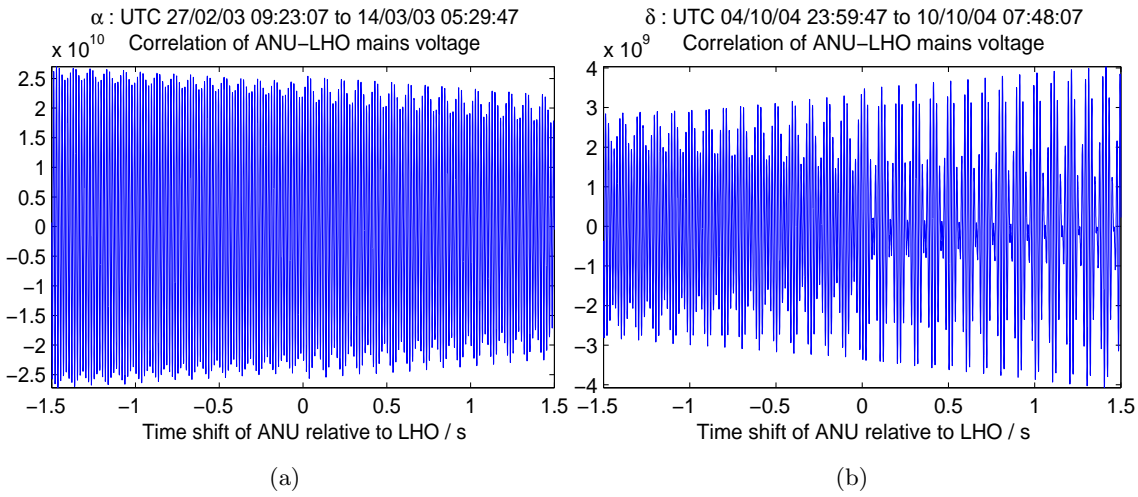


Figure 5.15: Correlation of ANU-LHO mains voltage monitors.

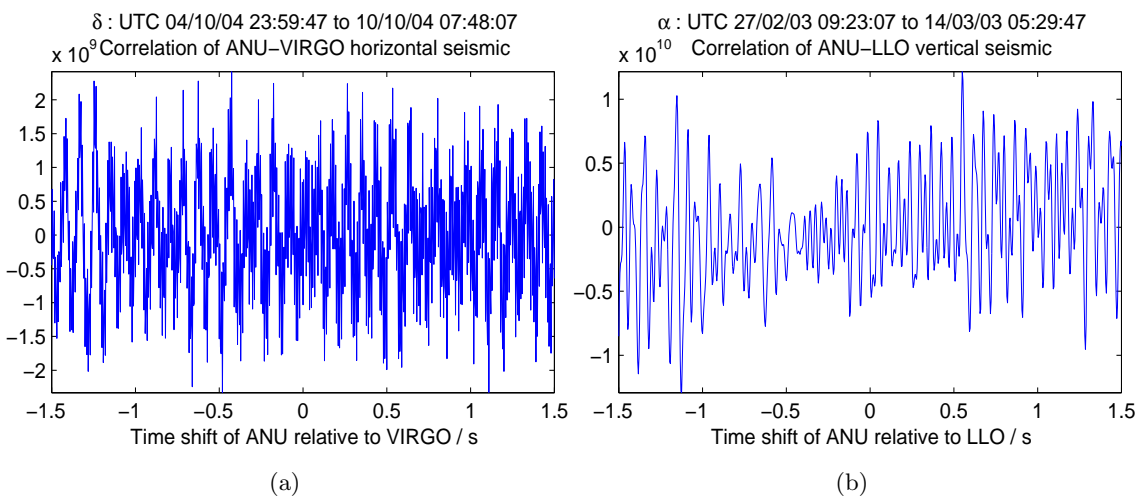


Figure 5.16: Correlation of the ANU-VIRGO horizontal and ANU-LLO vertical seismometers.

is also interesting to note that the general shapes of figures 5.17(a) and 5.17(b), replicated in figures 5.17(e) and 5.17(f), are instead replaced with a single general shape in figures 5.17(c) and 5.17(d). The similarity of both the horizontal and the vertical seismometers would appear to indicate a single influence during this stretch of data.

Figure 5.18 (page 61) shows the correlation of the LHO-LLO magnetic field sensors. We notice a linear slope enveloping the figure 5.18(a), which changes to a smaller slope in the opposite direction in figure 5.18(b), before reverting back to the original slope in figure 5.18(c). It is not clear what is implied by this linear envelope, but it would appear to corroborate figures 5.17 (page 60) with regard to some additional influence affecting LHO and LLO during the γ data set.

5.1.4 Coherence

Figure 5.19(a) and 5.19(b) (page 62) show minor (< 0.1) coherence of 50 Hz harmonics between ANU-VIRGO mains voltage monitors. It is interesting to note how the magnitudes of the coherence changes quite substantially between the two dataset; or this may simply indicate that the durations averaged over were not long enough to produce stable coherence, if it existed.

Figure 5.20 (page 63) shows more substantial coherence between ANU-LHO, ANU-LLO, and LHO-LLO magnetic field sensors. In particular, figure 5.21(e) shows substantial large coherence between LHO and LLO at a multitude of frequencies. Many of these originate from the 50 and 60 Hz harmonics of the mains. Other coherences are not as certain. In particular, we note a coherence line appearing at all three sites, at around 400 Hz, in the δ data run, with a magnitude of 0.18 in figure 5.20(b) (the maximum), a magnitude of 0.45 in figure 5.20(d) (also the maximum), and a magnitude of 0.16 in figure 5.21(b).

Figure 5.21 (page 64) shows coherences between LHO-LLO, LHO-VIRGO and LLO-VIRGO magnetic field sensors. Notice that the 400 Hz line is still present, with a magnitude of 0.16 in figure 5.21(d), and a magnitude of 0.5 in figure 5.21(f). Other large coherences at harmonics of 50 Hz, particularly at 600–1000 Hz are also present in a number of the figures. Although in the case of these coherences the 400 Hz line may just simply be another harmonic of 50 Hz, it is interesting that this line is so comparatively large in figures 5.20(b) and 5.20(d) (page 63), noting that 400 Hz is not a harmonic of 60 Hz, whereas no other 50 Hz lines appear in these figures to the same magnitude.

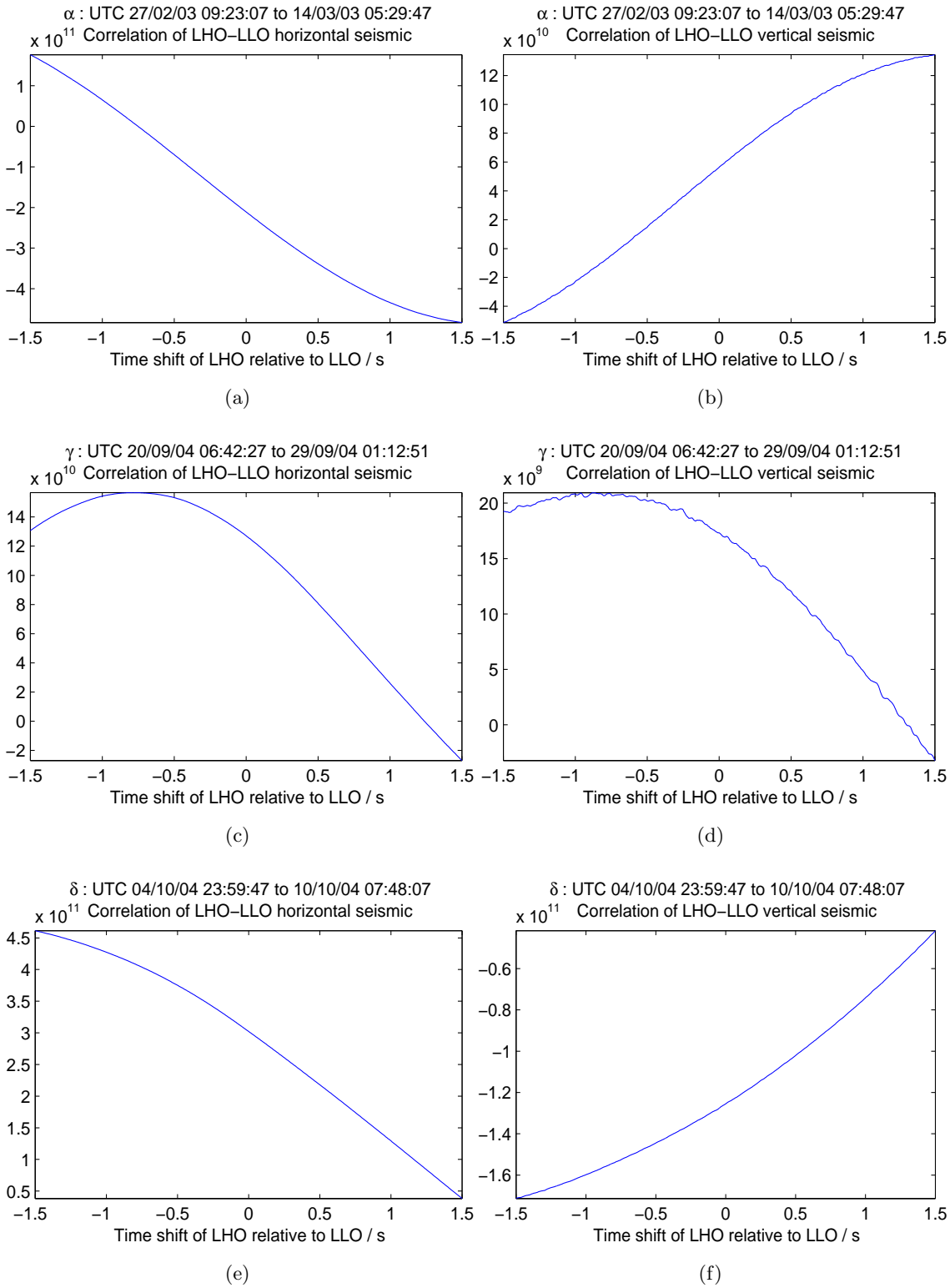
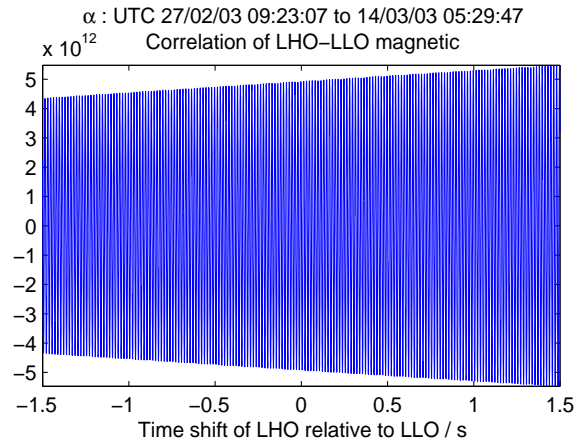
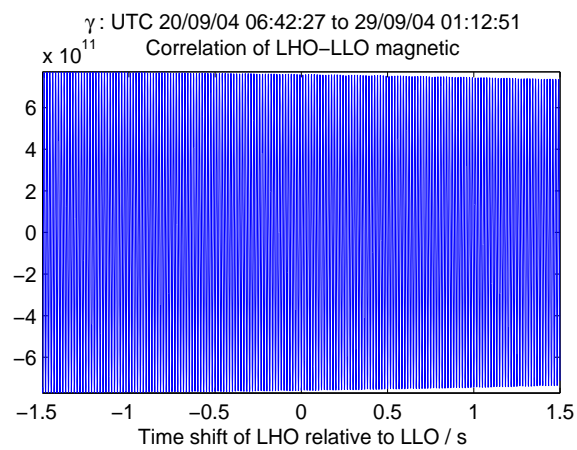


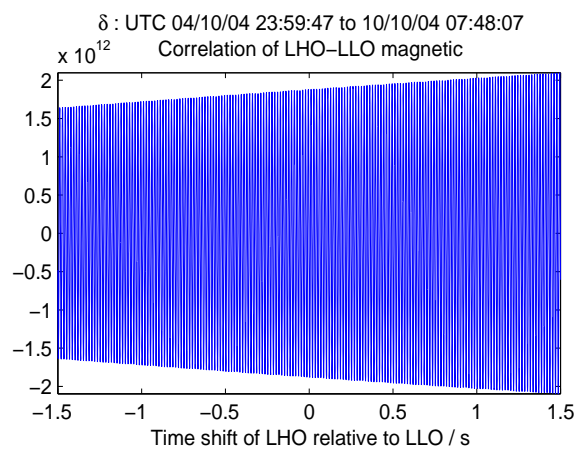
Figure 5.17: Correlation of the LHO-LLO horizontal and vertical seismometers.



(a)



(b)



(c)

Figure 5.18: Correlation of the LHO-LLO magnetic field sensors.

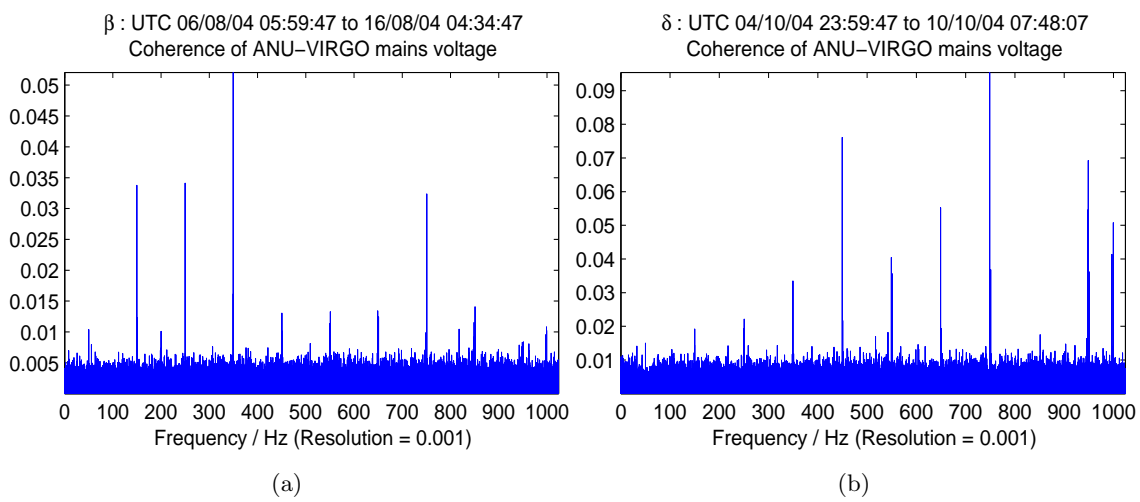


Figure 5.19: Coherence of the ANU-VIRGO mains voltage monitors.

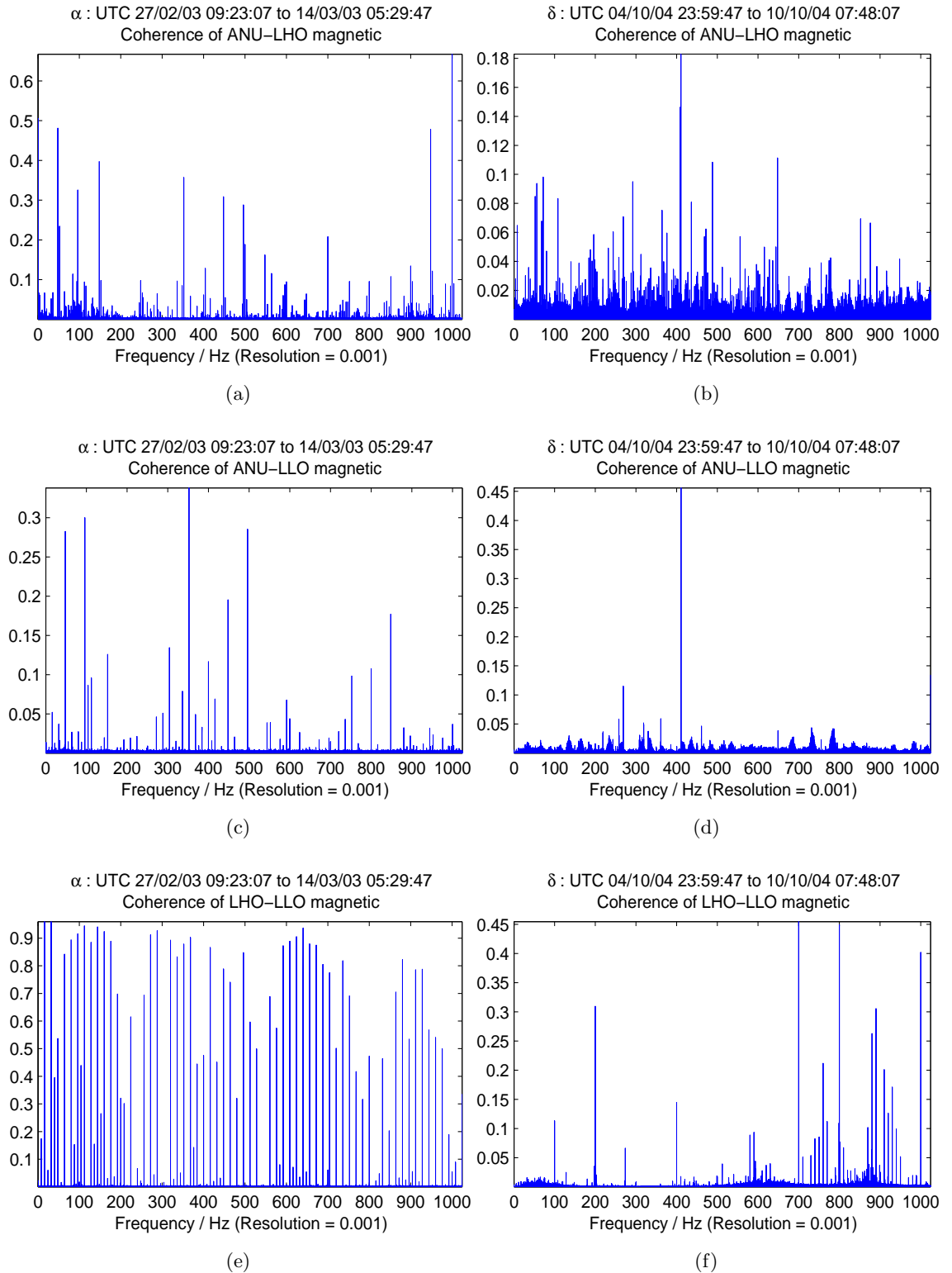


Figure 5.20: Coherence of the ANU-LHO, ANU-LLO and LHO-LLO magnetic field sensors.

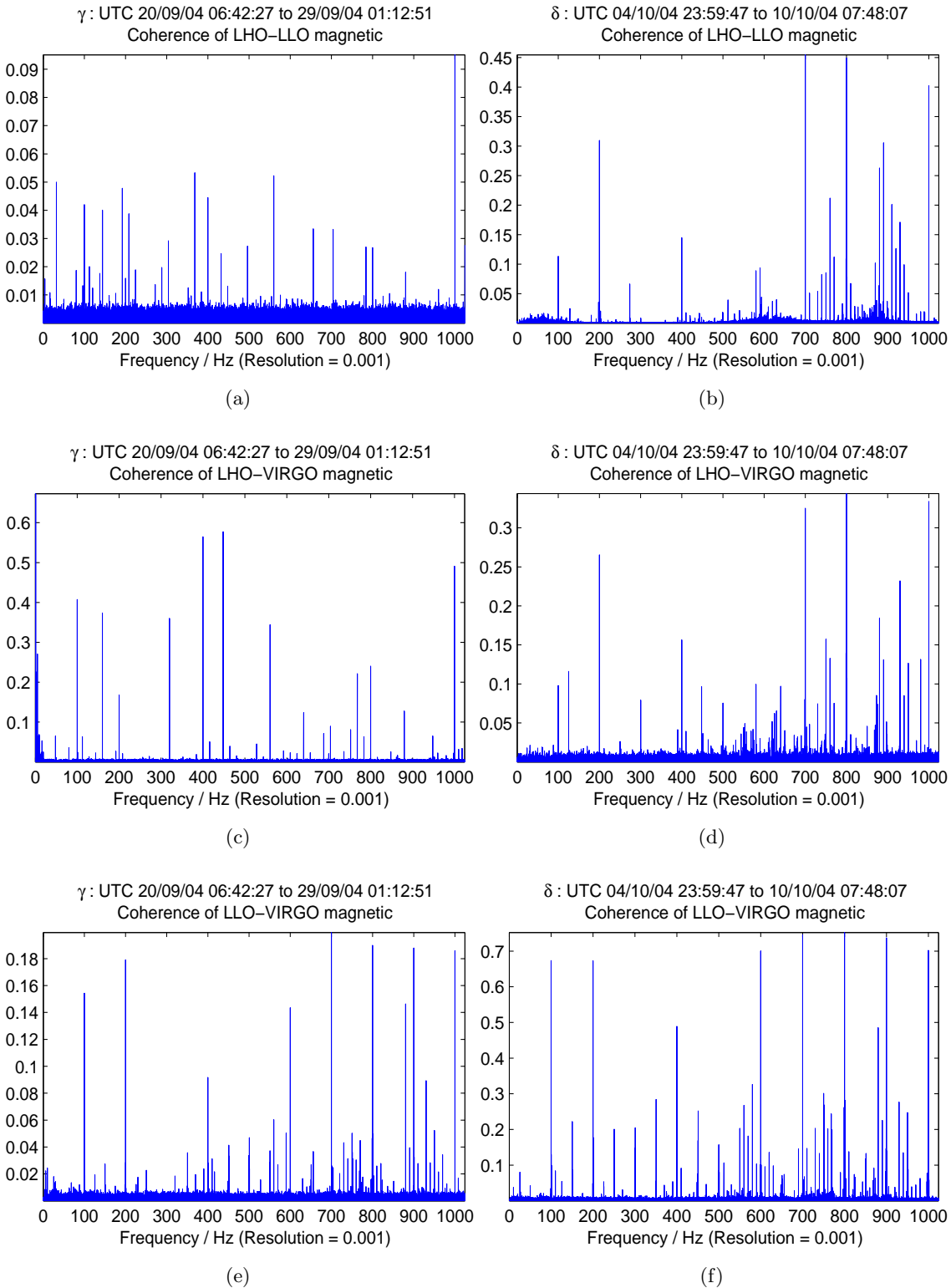


Figure 5.21: Coherence of the LHO-LLO, LHO-VIRGO and LLO-VIRGO magnetic field sensors.

5.2 Short timescale analyses

The search algorithm has produced spectrograms containing features which might be said to be interesting, if not significant. A representative selection of these spectrograms is shown in figures 5.22 (page 66), 5.23 (page 67), 5.24 (page 68), and 5.25 (page 69).

The most common transient event detected were bursts of activity covering most of the width of the spectrogram. They might also include other features. For example, figures 5.22(a) and 5.22(b) also contain a strong frequency line, spiralling into the burst in the first figure and flying out of it in the second. Sometimes a secondary burst was seen following the first (figure 5.22(h)). Bursts could also link between several strong frequency lines figures 5.22(e), 5.22(f)). Sometimes plumes of activity would remain for a short while after the burst (figures 5.23(g), 5.23(h)). As well as sharply defined bursts, momentary eruptions of harmonic lines were also observed (figure 5.23(e)). These were most often observed in the magnetic field sensor spectrograms (figures 5.24(c) and 5.24(d)). Sometimes the two would combine to produce sharp bursts at harmonic frequencies (figures 5.24(b), 5.24(e), and 5.24(f)).

Some other interesting features were found by the algorithm, including a few instances of wandering frequency lines (figures 5.24(g) and 5.24(h)). One of the more interesting features was bursts which appeared to delineate quantitatively different areas of the spectrogram – a spectrographic “phase change” (figure 5.25). They were found to delineate an area of either random or little activity from an area with some harmonic frequencies (figures 5.25(c), 5.25(a), 5.25(d) and 5.25(f)). They were found to also delineate between areas where the harmonic frequencies changed (figures 5.25(b), 5.25(g), and 5.25(h)).

In general, the simple difference-in-mean test performed as well as, if not better than, the remaining four statistical tests. The Student paired-t, the Wilcoxon signed rank and the Kolmogorov-Smirnov tests did not perform noticeably better than each other. The exception was the Wilcoxon-Mann-Whitney test, which performed very poorly. This was because its limited range resulted in it being much too sensitive to small perturbations which would then overwrite anything more important it might have found. The tests sometimes succeeded in detecting the same transient features (figures 5.25(g) and 5.25(h)).

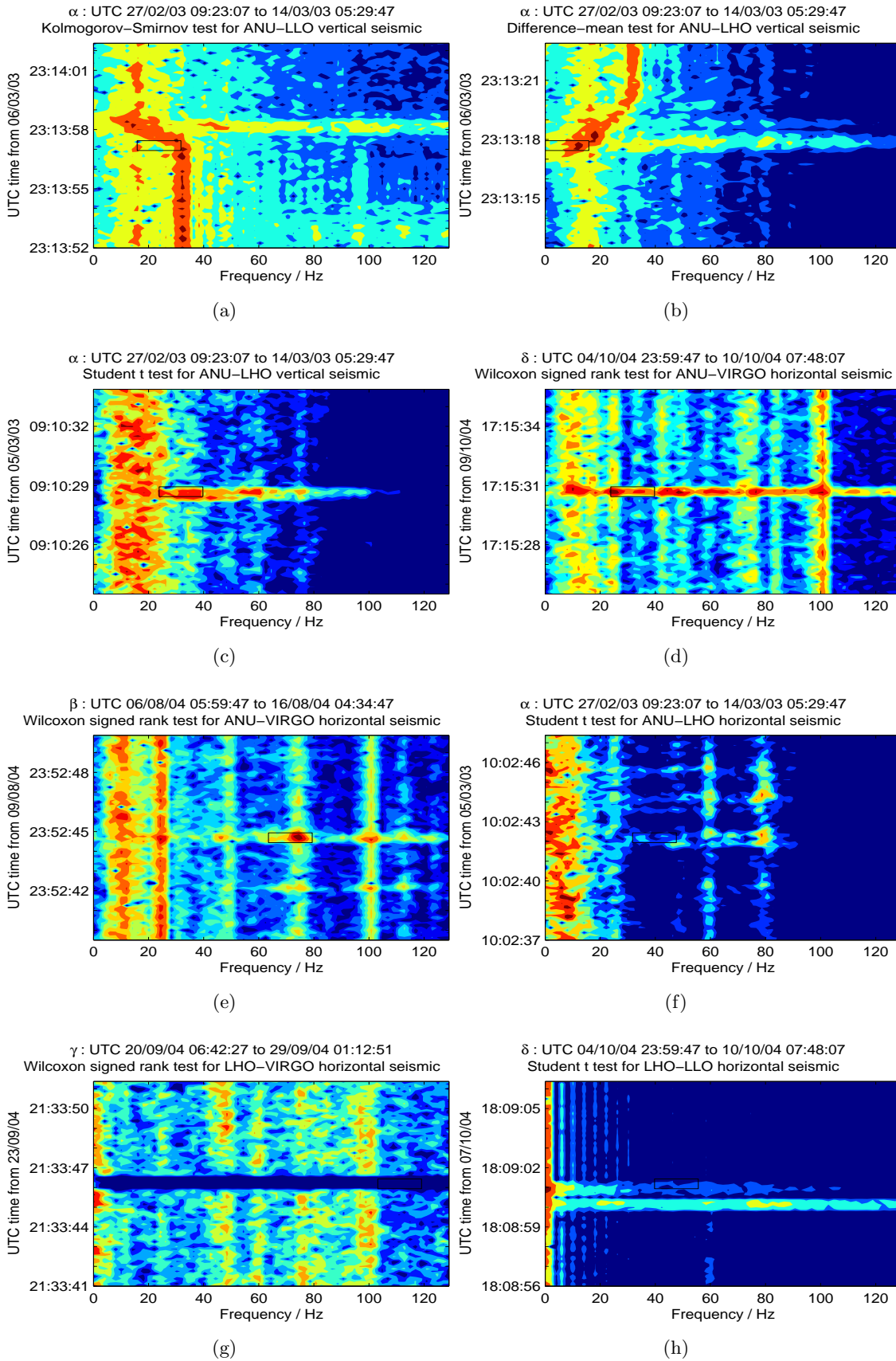


Figure 5.22: Bursts in seismometer spectrograms.

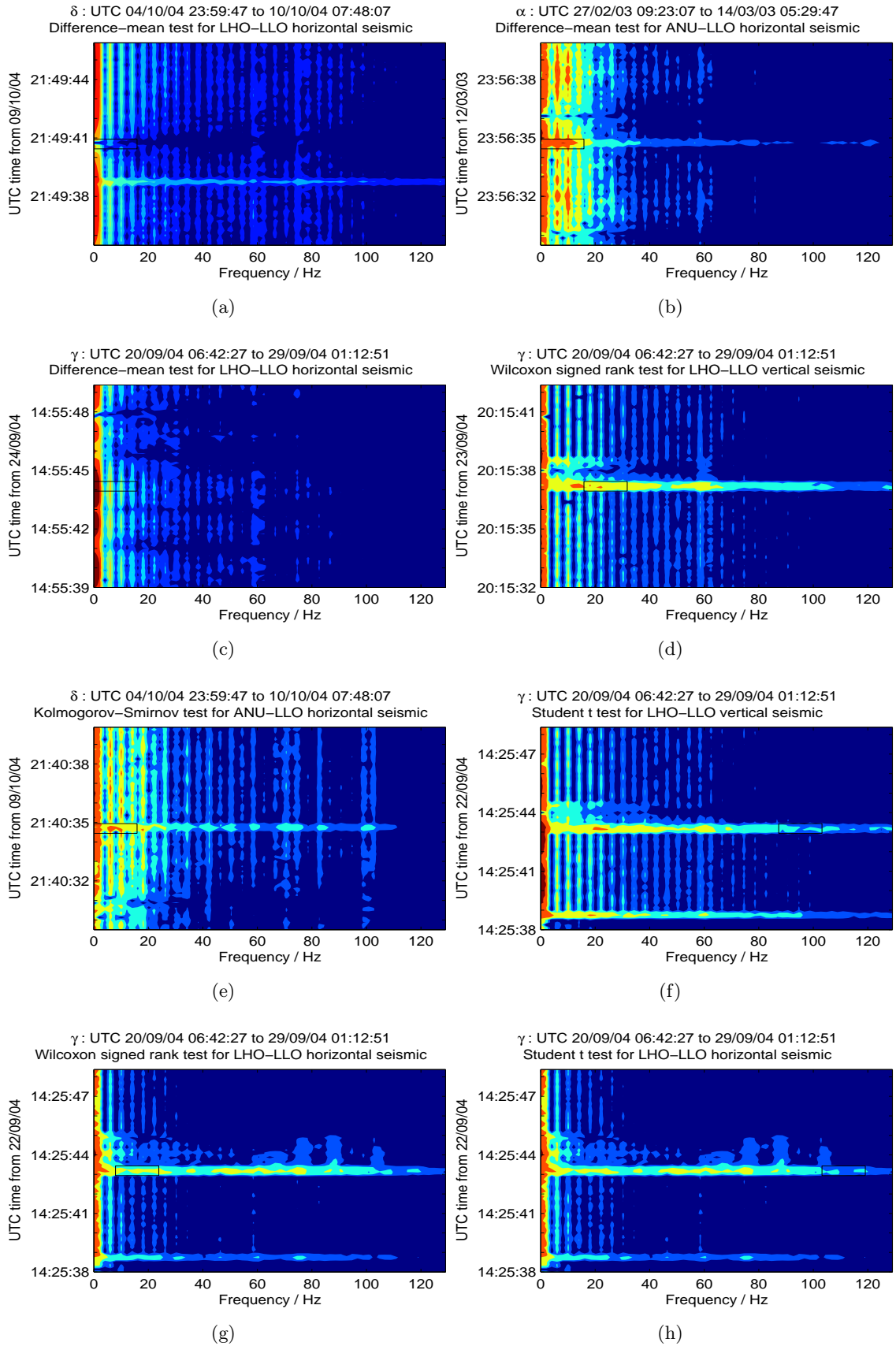


Figure 5.23: More bursts in seismometer spectrograms.

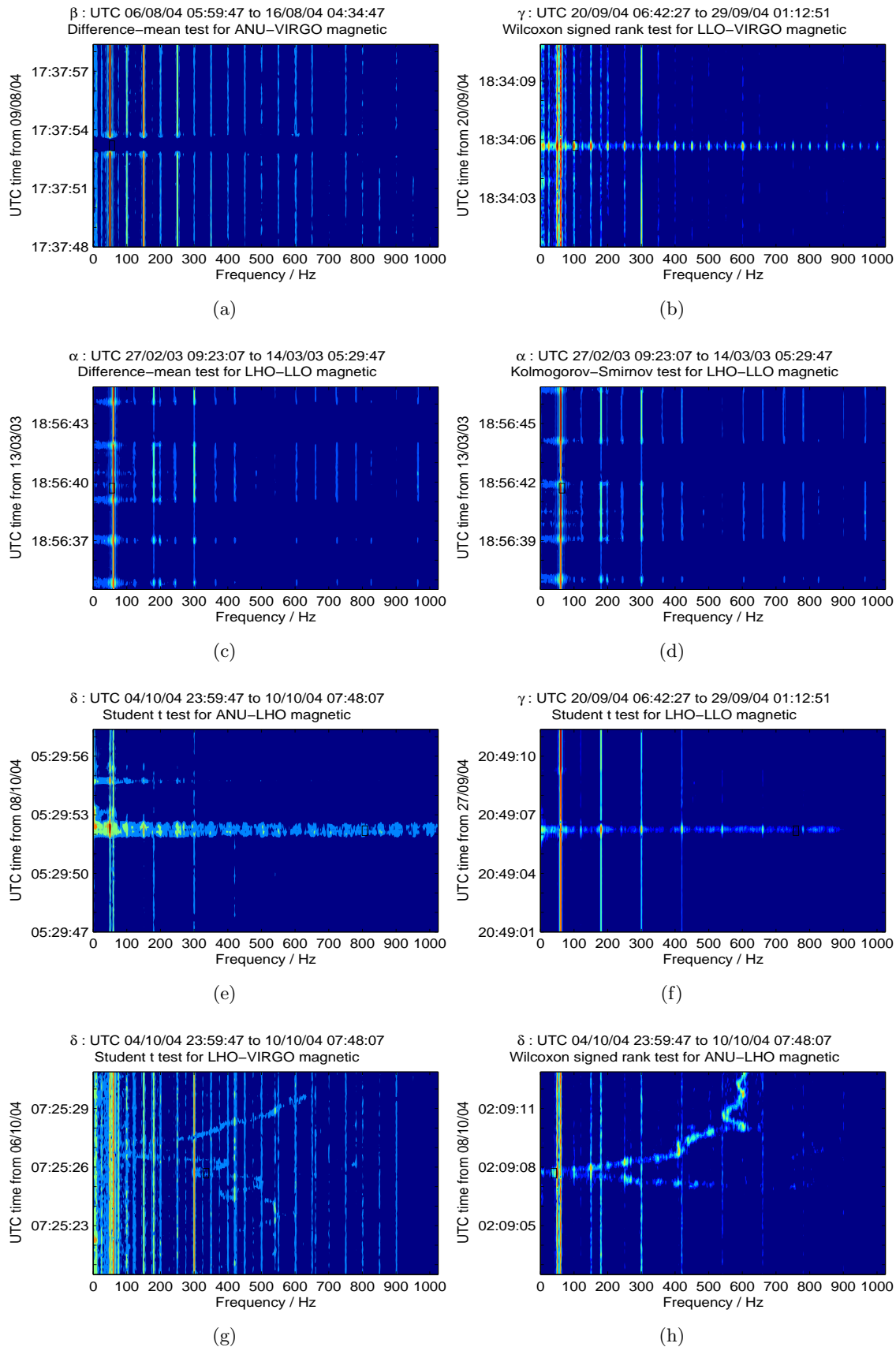


Figure 5.24: Bursts, gaps, and wandering lines in magnetic field sensor spectrograms.

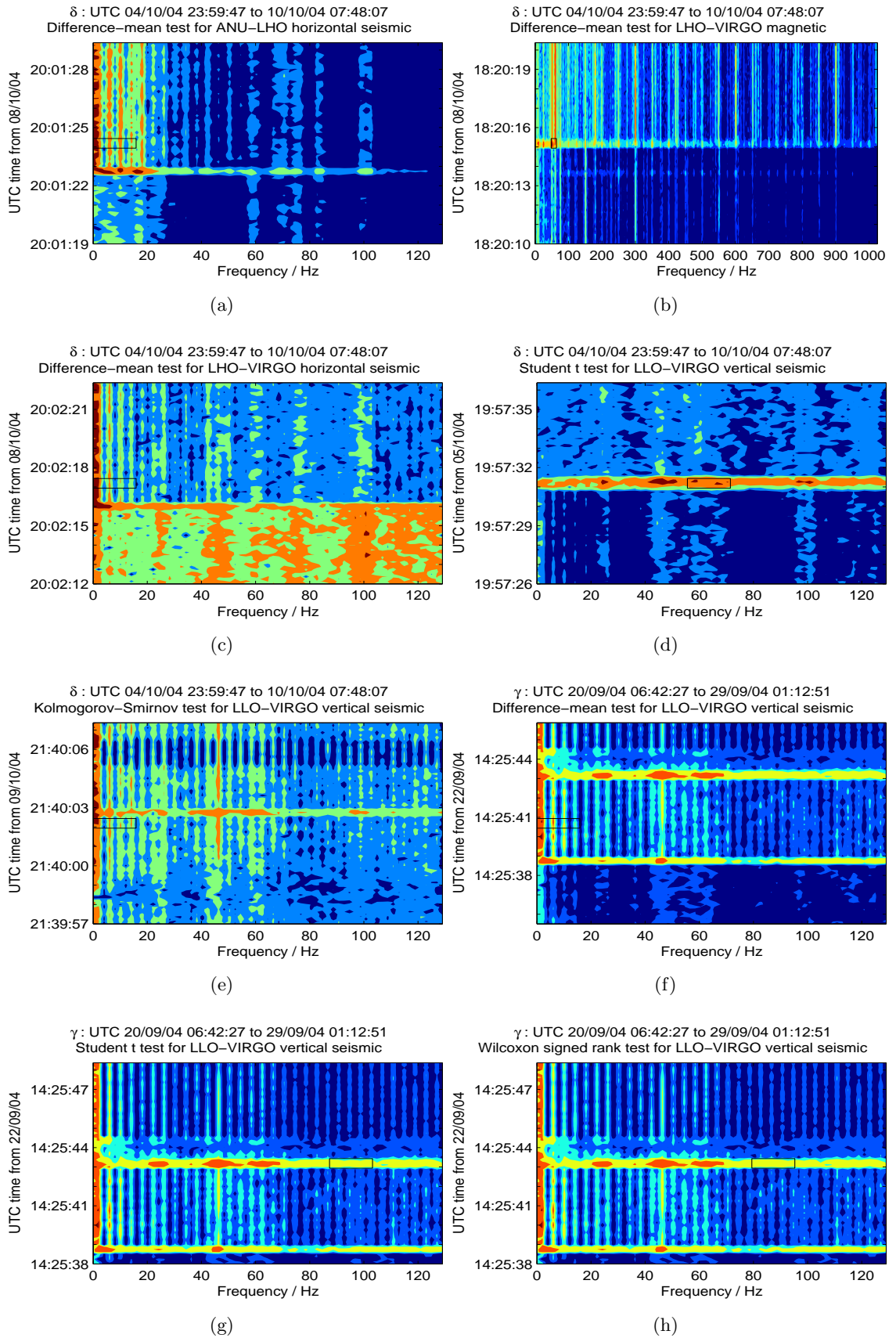


Figure 5.25: Spectrographic “phase changes” in seismometer and magnetic field sensor spectrograms.

Conclusion

In this project, the author made a substantial contribution to the physical environment monitoring station at The Australian National University. The data acquisition software was written in LabVIEW, and was extended in order to be able to write acquired data directly to the gravitational wave detector Frame file format. This greatly improved the simplicity and manageability of the monitoring station. The author also improved the organisation of previously archived data from the station.

A program was implemented for the purpose of processing and analysing data acquired from four physical environment monitoring stations. The program was written in C, and was able to be run as a parallel program across multiple computers by utilising an implementation of the MPI programming standard. The program implemented several well-defined algorithms used to identify long timescale correlations; for short timescale correlations, a transient event search algorithm was implemented to search spectrograms of the data. The algorithm used several statistical tests to try to characterise transient events. The program was eventually executed on several large data sets from the physical environment monitors.

The results of the analyses did not find any immediately significant long timescale correlations between the physical environment monitoring stations. Most candidate correlations could be initially explainable as being produced by independent but frequency-stable oscillators, such as the mains power grid. A deeper investigation would require considering the possible physical processes from which the observed correlations might have arisen; this was, however, beyond the scope of this project.

The search algorithm implemented to detect short timescale correlations was successful in distinguishing some interesting features. These correlations are not as easily explainable as some of the long timescale correlations, and are certainly worthy of further investigation.

There are many future directions in which this research might lead. Further development of the physical environment monitoring station and of the data acquisition software are still possible. The parallel processing program, while sufficient for the purposes of this project, has several deficiencies which might easily be improved upon to create a more professional data analysis software.

A deeper understanding of the possible physical processes which might give rise to global environment noise would be difficult, and most probably require further, more specific, monitoring data, but would, in the end, be of great service in the characterisation of global environment noise.

Further investigation into the nature of short timescale correlations and their possible causes would also be of great benefit. In particular, a comprehensive characterisation of short timescale correlations might enable the identification of those signals which are gravitational wave candidates. A search for short timescale correlations running parallel to

an operational gravitational wave detector could then identify these signals and eliminate them from consideration as gravitational wave candidates.

Much has been learned in this project. The author has acquired the skills to manage and analyse large data sets using parallel processing techniques. The software developed in the project will be useful as a base for future investigations. The results that were obtained, while not providing a definitive answer to the question of global environmental noise, certainly contain enough mysteries to be worthy of further and deeper investigation. It is the hope of the author that this project will provide a firm base upon which to continue such investigations.

Additional code

A.1 The parallel data processing program

A.1.1 The header

Listing A.1: The header

```
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include <math.h>
#include <unistd.h>
#include "mpi.h"
#include "rfftw.h"
#include "FrameL.h"
#include "Frv.h"
#ifdef FFTW_ENABLE_FLOAT
#define MPI_FFTW_REAL (MPI_FLOAT)
#else
#define MPI_FFTW_REAL (MPI_DOUBLE)
#endif
#define FFTW_PLAN_FLAGS (FFTW_MEASURE | FFTW_USE_WISDOM)
#define SWAP(a, b, t) t = a; a = b; b = t
typedef struct {
    char *string;
    int index;
} StringIndex;
typedef struct {
    int init;
    fftw_real *buffer;
    double start;
    double length;
} Buffer;
typedef struct {
    int FFT;
    fftw_real *fft_in;
    fftw_real *fft_out;
    fftw_real *fft_win;
```

```

    fftw_real fft_win_norm;
    int SPECT;
    fftw_real *spect;
    int CORR;
    fftw_real *corr;
    int AUTC;
    fftw_real *autc;
    rfftw_plan fft_plan;
} LongBuf;
typedef struct {
    intFFT;
    fftw_real *fft_in;
    fftw_real *fft_out;
    fftw_real *fft_win;
    fftw_real fft_win_norm;
    int SPECT;
    fftw_real *spect;
    rfftw_plan fft_plan;
} TrnsBuf;
typedef struct {
    double time;
    double stat;
    double ord;
} Stat;
const int CORRELATIONS = 2;
const int LINE = 1024;
const char TOUCH_FILENAME[ ] = "delete_me_to_stop";
const double SEND_SEGMENT = 100.0;
const int PROC_CONTINUE = -1;
const int PROC_STOP = -2;
const double LONG_SPECT_WINDOW = 500.0;
const int LONG_SPECT_OVERLAP = 2;
const double LONG_CORR_SHIFT = 1.5;
const double TRNS_SPECT_WINDOW = 0.125;
const int TRNS_SPECT_OVERLAP = 4;
const int SRCH_TIME_BIN = 4;
const int SRCH_TIME_OVERLAP = 1;
const int SRCH_FREQ_BIN = 4;
const int SRCH_FREQ_OVERLAP = 2;
const int SRCH_TIME_RANGE = 10;
const int STAT_SAVE = 10000;
const int STAT_SPCTGM_SAVE = 20;
const int STAT_COUNT = 5;
LongBuf *longbuf;
TrnsBuf *trnsbuf;
fftw_real *event_bin;
fftw_real **srch_bin;

```


A.1.2 The main subroutine

Listing A.2: The main subroutine variable declarations

```

int COLLECTORS, PROCESSORS, CROSSPROCESSORS, SIZE, i, j, retn, rank, size;
MPI_Group group;
MPI_Comm comm, local_comm;
int *coll_proc_ranks, coll_proc_max_rank, coll_proc_comm_i;
MPI_Group coll_proc_group;
MPI_Comm coll_proc_comm[COLRELATIONS];
FILE *fchannel;
char line[LINE], *token;
int CHANNELS, CHANNEL_NAMES;
double *sampling_rate;
StringIndex *channel_names;

```

Listing A.3: The main subroutine construction of the collector-to-processor/cross-processor communicators

```

coll_proc_ranks = (int*) malloc((2 + 2*(COLLECTORS-1)) * sizeof(int));
coll_proc_max_rank = COLLECTORS + PROCESSORS;
for (i = 0; i < COLRELATIONS; i++) {
    coll_proc_comm[i] = MPI_COMM_NULL;
}
coll_proc_comm_i = 0;
for (i = 0; i < COLLECTORS; i++) {
    coll_proc_ranks[0] = i;
    coll_proc_ranks[1] = COLLECTORS + i;
    for (j = 2; j < i+1; j++) {
        coll_proc_ranks[j]++;
    }
    for (j = i+2; j < COLLECTORS+1; j++) {
        coll_proc_ranks[j] = coll_proc_max_rank++;
    }
}
for (j = 0; j < COLLECTORS-1; j++) {
    coll_proc_ranks[2 + (COLLECTORS-1) + j] = CROSSPROCESSORS +
        coll_proc_ranks[2 + j];
}
comm = MPI_COMM_NULL;
MPI_Group_incl(group, 2 + 2*(COLLECTORS-1), coll_proc_ranks,
    &coll_proc_group);
MPI_Comm_create(MPI_COMM_WORLD, coll_proc_group, &comm);
MPI_Group_free(&coll_proc_group);
if (comm != MPI_COMM_NULL) {
    coll_proc_comm[coll_proc_comm_i++] = comm;
}
}
free(coll_proc_ranks);

```

Listing A.4: The main subroutine processing of the channels file

```

fchannel = fopen(argv[1], "r");
CHANNELS = 0;
sampling_rate = NULL;
CHANNEL_NAMES = 0;
channel_names = NULL;
while (fgets(line, LINE, fchannel) != NULL) {
    token = strtok(line, " \n");
    if (token != NULL) {
        sampling_rate = (double*) realloc(sampling_rate,
            (++CHANNELS)*sizeof(double));
        sampling_rate[CHANNELS-1] = strtod(token, NULL);
        token = strtok(NULL, " \n");
        while (token != NULL) {
            channel_names = (StringIndex*) realloc(channel_names,
                (++CHANNEL_NAMES)*sizeof(StringIndex));
            channel_names[CHANNEL_NAMES-1].string = (char*) malloc(strlen(token)+1);
            strcpy(channel_names[CHANNEL_NAMES-1].string, token);
            channel_names[CHANNEL_NAMES-1].index = CHANNELS-1;
            token = strtok(NULL, " \n");
        }
    }
}

```

Listing A.5: The main subroutine calls to the collector and processor subroutines

```

longInit(CHANNELS, sampling_rate);
trnsInit(CHANNELS, sampling_rate);
if (rank < COLLECTORS) {
    if (coll_proc_comm_i == 1) {
        retn = collector(local_comm, coll_proc_comm[coll_proc_comm_i-1], CHANNELS,
            sampling_rate, CHANNEL_NAMES, channel_names, argv[rank+2]);
    }
    else {
        return error(rank, rank, "Unexpected number of communicators");
    }
}
else {
    if (rank < COLLECTORS + PROCESSORS + CROSSPROCESSORS) {
        retn = processor(coll_proc_comm_i, coll_proc_comm, 1, CHANNELS, sampling_rate);
    }
    else {
        retn = processor(coll_proc_comm_i, coll_proc_comm, 2, CHANNELS, sampling_rate);
    }
}
longFinal(CHANNELS);
trnsFinal(CHANNELS);

```

Listing A.6: The main subroutine cleanup

```

if (local_comm != MPI_COMM_NULL) {
    MPI_Comm_free(&local_comm);
}
for (i = 0; i < coll_proc_comm_i; i++) {
    if (coll_proc_comm[i] != MPI_COMM_NULL) {
        MPI_Comm_free(&coll_proc_comm[i]);
    }
}
free(sampling_rate);
for (i = 0; i < CHANNEL_NAMES; i++) {
    free(channel_names[i].string);
}
free(channel_names);
MPI_Group_free(&group);
MPI_Finalize();
return retn;
}

```

A.1.3 The collector subroutine

Listing A.7: The collector subroutine variable declarations

```

int i, j, local_rank, local_size;
char *listout_filename, line[LINE];
FILE *flist, FILE *flistout;
FrFile *frfile;
FrameH *frame;
FrRawData *rawData;
FrAdcData *adcData;
FrVect *vectData;
fftw_real *vectdataData;
Buffer series[CHANNELS];
double start_sync[CHANNELS], double start_sync_max[CHANNELS];
int done_start_sync, all_have_data, coll_status, coll_continue;

```

Listing A.8: The collector subroutine initialisation

```

for (i = 0; i < CHANNELS; i++) {
    bufferInit(&series[i]);
}
done_start_sync = 0;
listout_filename = sprintfalloc("%s.out", list_filename);
fclose(fopen(listout_filename, "w"));
if ((rawData = frame->rawData) != NULL) {
    for (adcData = rawData->firstAdc; adcData != NULL; adcData =
        adcData->next) {
        for (i = 0; i < CHANNEL_NAMES; i++) {
            if (strstr(adcData->name, channel_names[i].string) != NULL) {
                j = channel_names[i].index;
            }
        }
    }
}

```

```

#ifdef FFTW_ENABLE_FLOAT
    vectData = FrvCopyToF(adcData->data, 1.0, NULL);
    vectdataData = (fftw_real*) vectData->dataF;
#else
    vectData = FrvCopyToD(adcData->data, 1.0, NULL);
    vectdataData = (fftw_real*) vectData->dataD;
#endif

    FrvZeroMean(vectData, NULL);
    bufferAdd(&series[j], sampling_rate[j], vectdataData, round(((double)
        frame->GTimeS) + ((double) frame->GTimeN) / 1e9),
        vectData->nData / adcData->sampleRate,
        adcData->sampleRate);
    FrVectFree(vectData);
    vectdataData = NULL;
}
}
}
}

```

Listing A.9: The collector subroutine synchronisation of starting times

```

if (done_start_sync == 0) {
    for (i = 0; i < CHANNELS; i++) {
        start_sync[i] = series[i].start;
        start_sync_max[i] = 0;
    }
    MPI_Allreduce(start_sync, start_sync_max, CHANNELS, MPI_DOUBLE,
        MPI_MAX, local_comm);
    for (i = 0; i < CHANNELS; i++) {
        bufferShift(&series[i], sampling_rate[i], start_sync_max[i]);
        if (local_rank == 0) {
            printf("Started analysis of channel %i at time %0.0f\n", i,
                start_sync_max[i]);
            fflush(stdout);
        }
    }
}
done_start_sync = 1;
}

```

Listing A.10: The collector subroutine transmission of data

```

if (coll_continue) {
    for (i = 0; i < CHANNELS; i++) {
        MPI_Bcast(&i, 1, MPI_INT, 0, to_proc_comm);
        MPI_Bcast(&series[i].start, 1, MPI_DOUBLE, 0, to_proc_comm);
        MPI_Bcast(series[i].buffer, SEND_SEGMENT * sampling_rate[i],
            MPI_FFTW_REAL, 0, to_proc_comm);
        bufferShift(&series[i], sampling_rate[i], series[i].start +
            SEND_SEGMENT);
    }
}

```

```

    }
    MPI_Bcast((int*) &PROC_CONTINUE, 1, MPI_INT, 0, to_proc_comm);
  }
}
}

```

Listing A.11: The collector subroutine cleanup

```

}
FrFileEnd(frfile);
flistout = fopen(listout_filename, "a");
fprintf(flistout, "%s\n", line);
fclose(flistout);
}
fclose(flist);
if (coll_continue) {
  coll_status = 0; coll_continue = 0;
  MPI_Allreduce(&coll_status, &coll_continue, 1, MPI_INT, MPI_LAND, local_comm);
}
MPI_Bcast((int*) &PROC_STOP, 1, MPI_INT, 0, to_proc_comm);
if (local_rank == 0) {
  for (i = 0; i < CHANNELS; i++) {
    printf("Stopped analysis of channel %i at time %0.0f\n", i, series[i].start
          + series[i].length);
    fflush(stdout);
  }
}
for (i = 0; i < CHANNELS; i++) {
  bufferFree(&series[i]);
}
return EXIT_SUCCESS;
}

```

A.1.4 The processor subroutine

Listing A.12: The processor subroutine variable declarations

```

int i, j;
char line[LINE], proc_filename_id[LINE];
int from_coll_comm_stopped[FROM_COLL_COMM], from_coll_comm_left;
fftw_real *receive[CHANNELS];
double receive_start[CHANNELS];
Buffer long_series[CHANNELS][FROM_COLL_COMM];
Buffer corr_series[CHANNELS][FROM_COLL_COMM];
Buffer trns_series[CHANNELS][FROM_COLL_COMM];
Buffer spctgm[CHANNELS];
double spctgm_sampling_rate[CHANNELS];

```

Listing A.13: The processor subroutine initialisation

```

proc_filename_id[0] = '\0';

```

```

for (j = 0; j < FROM_COLL_COMM; j++) {
    MPI_Bcast(line, LINE, MPI_CHAR, 0, from_coll_comm[j]);
    if (j > 0) {
        strcat(proc_filename_id, "-");
    }
    strcat(proc_filename_id, line);
}
for (i = 0; i < CHANNELS; i++) {
    receive[i] = (fftw_real*) malloc(SEND_SEGMENT * sampling_rate[i] *
        sizeof(fftw_real));
    receive_start[i] = 0;
    for (j = 0; j < FROM_COLL_COMM; j++) {
        bufferInit(&long_series[i][j]);
        bufferInit(&corr_series[i][j]);
        bufferInit(&trns_series[i][j]);
    }
    bufferInit(&spctgm[i]);
    spctgm_sampling_rate[i] = 0;
}
for (i = 0; i < FROM_COLL_COMM; i++) {
    from_coll_comm_stopped[i] = 0;
}
from_coll_comm_left = FROM_COLL_COMM;

```

Listing A.14: The processor subroutine reception of data

```

if (i > PROC_CONTINUE) {
    MPI_Bcast(&receive_start[i], 1, MPI_DOUBLE, 0, from_coll_comm[j]);
    MPI_Bcast(receive[i], SEND_SEGMENT * sampling_rate[i],
        MPI_FFTW_REAL, 0, from_coll_comm[j]);
    if ((FROM_COLL_COMM == 1) | (crossproc_set == 1)) {
        bufferAdd(&long_series[i][j], sampling_rate[i], receive[i], receive_start[i],
            SEND_SEGMENT, sampling_rate[i]);
        bufferAdd(&corr_series[i][j], sampling_rate[i], receive[i], receive_start[i],
            SEND_SEGMENT, sampling_rate[i]);
    }
    else {
        bufferAdd(&trns_series[i][j], sampling_rate[i], receive[i], receive_start[i],
            SEND_SEGMENT, sampling_rate[i]);
    }
}

```

Listing A.15: The processor subroutine execution of the analyses

```

if (from_coll_comm_left == FROM_COLL_COMM) {
    for (i = 0; i < CHANNELS; i++) {
        if (FROM_COLL_COMM == 1) {
            doLongPower(&long_series[i][0], sampling_rate[i], i, proc_filename_id);
            doLongAuto(&corr_series[i][0], sampling_rate[i], i, proc_filename_id);
        }
    }
}

```

```

else {
    if (crossproc_set == 1) {
        doLongCross(&long_series[i][0], &long_series[i][1], sampling_rate[i], i,
                    proc_filename_id);
        doLongCorr(&corr_series[i][0], &corr_series[i][1], sampling_rate[i], i,
                  proc_filename_id);
    }
    else {
        doTrnsCross(&trns_series[i][0], &trns_series[i][1], sampling_rate[i], &spctgm[i],
                   &spctgm_sampling_rate[i], i);
        doTrnsSearch(&spctgm[i], spctgm_sampling_rate[i], sampling_rate[i], i,
                    proc_filename_id);
    }
}
}
}
}

```

Listing A.16: The processor subroutine cleanup

```

}
for (i = 0; i < CHANNELS; i++) {
    free(receive[i]);
    for (j = 0; j < FROM_COLL_COMM; j++) {
        bufferFree(&long_series[i][j]);
        bufferFree(&corr_series[i][j]);
        bufferFree(&trns_series[i][j]);
    }
    bufferFree(&spctgm[i]);
}
return EXIT_SUCCESS;
}

```

A.1.5 Implementation of the long timescale analyses

Listing A.17: The doLongAuto subroutine

```

void doLongAuto(Buffer *series, double sampling_rate, int channel, char *filename_id) {
    int i, k;
    double length;
    char *auto_filename;
    auto_filename = sprintfalloc("long_auto_%s-%i", filename_id, channel);
    if (algrRead(auto_filename, longbuf[channel].AUTC, sizeof(longbuf[channel].autc[0]),
                longbuf[channel].autc, "Sampling: %f\nShift: %f\n", sampling_rate,
                LONG_CORR_SHIFT)) {
        for (i = 0; i < longbuf[channel].AUTC; i++) {
            longbuf[channel].autc[i] = 0;
        }
    }
    while ((length = series->length - LONG_CORR_SHIFT) > 0) {
        for (i = 0; i <= LONG_CORR_SHIFT * sampling_rate; i++) {

```

```

    for (k = 0; k < length * sampling_rate; k++) {
        longbuf[channel].autc[i] += series->buffer[k] * series->buffer[k + i];
    }
}
bufferShift(series, sampling_rate, series->start + length);
}
algrWrite(auto_filename, longbuf[channel].AUTC, sizeof(longbuf[channel].autc[0]),
    longbuf[channel].autc, "");
free(auto_filename);
return;
}

```

Listing A.18: The doLongCorr subroutine

```

void doLongCorr(Buffer *series1, Buffer *series2, double sampling_rate, int channel, char
    *filename_id) {
    int i, j, k;
    double length;
    char *corr_filename;
    corr_filename = sprintfalloc("long_corr_%s-%i", filename_id, channel);
    if (algrRead(corr_filename, longbuf[channel].CORR, sizeof(longbuf[channel].corr[0]),
        longbuf[channel].corr, "Sampling: %f\nShift: %f\n", sampling_rate,
        LONG_CORR_SHIFT)) {
        for (i = 0; i < longbuf[channel].CORR; i++) {
            longbuf[channel].corr[i] = 0;
        }
    }
    while ((length = ((series1->length < series2->length) ? series1->length :
        series2->length) - LONG_CORR_SHIFT) > 0) {
        for (i = 0, j = LONG_CORR_SHIFT * sampling_rate; j > 0; i++, j--) {
            for (k = 0; k < length * sampling_rate; k++) {
                longbuf[channel].corr[i] += series1->buffer[k] * series2->buffer[k + j];
            }
        }
        for (j = 0; j <= LONG_CORR_SHIFT * sampling_rate; i++, j++) {
            for (k = 0; k < length * sampling_rate; k++) {
                longbuf[channel].corr[i] += series1->buffer[k + j] * series2->buffer[k];
            }
        }
        bufferShift(series1, sampling_rate, series1->start + length);
        bufferShift(series2, sampling_rate, series2->start + length);
    }
    algrWrite(corr_filename, longbuf[channel].CORR, sizeof(longbuf[channel].corr[0]),
        longbuf[channel].corr, "");
    free(corr_filename);
    return;
}

```

Note: The reader may notice that, from the first **for** loop to the second, the independent variable of the correlation j , which is equivalently τ in equation 3.5, has switched position

from series2 to series1. This is because each Buffer only extends in one direction from its initial data point, whereas the correlation extends equally in both directions. Although this would seem to imply a change in the definition of the correlation, it actually only implies that the values of the correlation computed in one sweep do not all correspond to exactly the same time τ . Since the correlation will be calculated over long data sets, this effect reduces to insignificance.

Listing A.19: The doLongPower subroutine

```

void doLongPower(Buffer *series, double sampling_rate, int channel, char *filename_id) {
    int i;
    char *power_filename;
    int add_power;
    power_filename = sprintfalloc("long_power_%s-%i", filename_id, channel);
    add_power = !algrRead(power_filename, longbuf[channel].SPECT,
        sizeof(longbuf[channel].spect[0]), longbuf[channel].spect, "Sampling:
        %f\nLength: %f\n", sampling_rate, LONG_SPECT_WINDOW *
        LONG_SPECT_OVERLAP);
    while (series->length >= LONG_SPECT_WINDOW * LONG_SPECT_OVERLAP) {
        for (i = 0; i < longbuf[channel].FFT; i++) {
            longbuf[channel].fft_in[i] = series->buffer[i] * longbuf[channel].fft_win[i];
        }
        rfftw(longbuf[channel].fft_plan, 1, longbuf[channel].fft_in, 1, longbuf[channel].FFT,
            longbuf[channel].fft_out, 1, longbuf[channel].FFT);
        algrCross(longbuf[channel].FFT, longbuf[channel].fft_out, longbuf[channel].fft_out,
            longbuf[channel].spect, longbuf[channel].fft_win_norm, add_power);
        bufferShift(series, sampling_rate, series->start + LONG_SPECT_WINDOW);
    }
    algrWrite(power_filename, longbuf[channel].SPECT, sizeof(longbuf[channel].spect[0]),
        longbuf[channel].spect, "");
    free(power_filename);
    return;
}

```

Listing A.20: The doLongCross subroutine

```

void doLongCross(Buffer *series1, Buffer *series2, double sampling_rate, int channel,
    char *filename_id) {
    int i;
    char *cross_filename;
    int add_cross;
    cross_filename = sprintfalloc("long_cross_%s-%i", filename_id, channel);
    add_cross = !algrRead(cross_filename, 2 * longbuf[channel].SPECT,
        sizeof(longbuf[channel].spect[0]), longbuf[channel].spect, "Sampling:
        %f\nLength: %f\n", sampling_rate, LONG_SPECT_WINDOW *
        LONG_SPECT_OVERLAP);
    while ((series1->length >= LONG_SPECT_WINDOW * LONG_SPECT_OVERLAP) &
        (series2->length >= LONG_SPECT_WINDOW *
        LONG_SPECT_OVERLAP)) {
        for (i = 0; i < longbuf[channel].FFT; i++) {

```

```

    longbuf[channel].fft_in[i] = series1->buffer[i] * longbuf[channel].fft_win[i];
    longbuf[channel].fft_in[longbuf[channel].FFT+i] = series2->buffer[i] *
        longbuf[channel].fft_win[i];
}
rfftw(longbuf[channel].fft_plan, 2, longbuf[channel].fft_in, 1, longbuf[channel].FFT,
    longbuf[channel].fft_out, 1, longbuf[channel].FFT);
algrCross(longbuf[channel].FFT, &longbuf[channel].fft_out[0],
    &longbuf[channel].fft_out[longbuf[channel].FFT], longbuf[channel].spect,
    longbuf[channel].fft_win_norm, add_cross);
bufferShift(series1, sampling_rate, series1->start + LONG_SPECT_WINDOW);
bufferShift(series2, sampling_rate, series2->start + LONG_SPECT_WINDOW);
}
algrWrite(cross_filename, 2 * longbuf[channel].SPECT,
    sizeof(longbuf[channel].spect[0]), longbuf[channel].spect, "");
free(cross_filename);
return;
}

```

A.1.6 Implementation of the short timescale analyses

Listing A.21: The doTrnsPower subroutine

```

void doTrnsPower(Buffer *series, double series_sampling_rate, Buffer *spctgm, double
    *spctgm_sampling_rate, int channel) {
    int i;
    while (series->length >= TRNS_SPECT_WINDOW * TRNS_SPECT_OVERLAP) {
        for (i = 0; i < trnsbuf[channel].FFT; i++) {
            trnsbuf[channel].fft_in[i] = series->buffer[i] * trnsbuf[channel].fft_win[i];
        }
        rfftw(trnsbuf[channel].fft_plan, 1, trnsbuf[channel].fft_in, 1, trnsbuf[channel].FFT,
            trnsbuf[channel].fft_out, 1, trnsbuf[channel].FFT);
        algrCross(trnsbuf[channel].FFT, trnsbuf[channel].fft_out, trnsbuf[channel].fft_out,
            trnsbuf[channel].spect, trnsbuf[channel].fft_win_norm, 0);
        if (*spctgm_sampling_rate == 0) {
            *spctgm_sampling_rate = trnsbuf[channel].SPECT / TRNS_SPECT_WINDOW;
        }
        bufferAdd(spctgm, *spctgm_sampling_rate, trnsbuf[channel].spect, series->start,
            trnsbuf[channel].SPECT / * spctgm_sampling_rate, *spctgm_sampling_rate);
        bufferShift(series, series_sampling_rate, series->start + TRNS_SPECT_WINDOW);
    }
    return;
}

```

Listing A.22: The doTrnsCross subroutine

```

void doTrnsCross(Buffer *series1, Buffer *series2, double series_sampling_rate, Buffer
    *spctgm, double * spctgm_sampling_rate, int channel) {
    int i;
    while ((series1->length >= TRNS_SPECT_WINDOW * TRNS_SPECT_OVERLAP) &
        (series2->length >= TRNS_SPECT_WINDOW * TRNS_SPECT_OVERLAP))
        {

```

```

for (i = 0; i < trnsbuf[channel].FFT; i++) {
    trnsbuf[channel].fft_in[i] = series1->buffer[i] * trnsbuf[channel].fft_win[i];
    trnsbuf[channel].fft_in[trnsbuf[channel].FFT+i] = series2->buffer[i] *
        trnsbuf[channel].fft_win[i];
}
rfftw(trnsbuf[channel].fft_plan, 2, trnsbuf[channel].fft_in, 1, trnsbuf[channel].FFT,
    trnsbuf[channel].fft_out, 1, trnsbuf[channel].FFT);
algrCrossSqr(trnsbuf[channel].FFT, &trnsbuf[channel].fft_out[0],
    &trnsbuf[channel].fft_out[trnsbuf[channel].FFT], trnsbuf[channel].spect,
    trnsbuf[channel].fft_win_norm, 0);
if (*spctgm_sampling_rate == 0) {
    *spctgm_sampling_rate = trnsbuf[channel].SPECT / TRNS_SPECT_WINDOW;
}
bufferAdd(spctgm, *spctgm_sampling_rate, trnsbuf[channel].spect, series1->start,
    trnsbuf[channel].SPECT / * spctgm_sampling_rate, *spctgm_sampling_rate);
bufferShift(series1, series_sampling_rate, series1->start + TRNS_SPECT_WINDOW);
bufferShift(series2, series_sampling_rate, series2->start + TRNS_SPECT_WINDOW);
}
return;
}

```

Listing A.23: The doTrnsSearch subroutine

```

void doTrnsSearch(Buffer *spctgm, double spctgm_sampling_rate, double sampling_rate,
    int channel, char *filename_id) {
    int SRCH_TIME_RANGE_TOTAL = (2 * SRCH_TIME_RANGE) + 1;
    int SRCH_TIME_LENGTH = SRCH_TIME_BIN * (SRCH_TIME_OVERLAP +
        SRCH_TIME_RANGE_TOTAL - 1);
    int SRCH_FREQ_COUNT = floor((trnsbuf[channel].SPECT / SRCH_FREQ_BIN) -
        SRCH_FREQ_OVERLAP + 1);
    int f, t, i, j;
    fftw_real *bin;
    double event_stat[STAT_COUNT];
    int event_ord;
    Stat stat_list[STAT_COUNT][STAT_SAVE], swap;
    char *stat_list_filename[STAT_COUNT], *spctgm_filename;
    for (i = 0; i < STAT_COUNT; i++) {
        stat_list_filename[i] = sprintfalloc("trns_stat-%i_%s-%i", i, filename_id, channel);
        if (algrRead(stat_list_filename[i], STAT_SAVE, sizeof(stat_list[i][0]), stat_list[i], "")) {
            for (j = 0; j < STAT_SAVE; j++) {
                stat_list[i][j].time = 0;
                stat_list[i][j].stat = 0;
                stat_list[i][j].ord = j;
            }
        }
    }
    while (spctgm->length >= TRNS_SPECT_WINDOW * SRCH_TIME_LENGTH) {
        for (f = 0; f < SRCH_FREQ_COUNT; f++) {
            for (t = 0; t < SRCH_TIME_RANGE_TOTAL; t++) {

```

```

if (t < SRCH_TIME_RANGE) {
    bin = srch_bin[t];
}
else if (t < SRCH_TIME_RANGE + 1) {
    bin = event_bin;
}
else {
    bin = srch_bin[t-1];
}
for (i = 0; i < SRCH_TIME_BIN * SRCH_TIME_OVERLAP; i++) {
    for (j = 0; j < SRCH_FREQ_BIN * SRCH_FREQ_OVERLAP; j++) {
        bin[(i * SRCH_FREQ_BIN * SRCH_FREQ_OVERLAP) + j] =
            spctgm->buffer[((t * SRCH_TIME_BIN) + i) * trnsbuf[channel].SPECT +
                (f * SRCH_FREQ_BIN) + j];
    }
}
}
statCompute(SRCH_TIME_BIN * SRCH_TIME_OVERLAP * SRCH_FREQ_BIN *
    SRCH_FREQ_OVERLAP, 2 * SRCH_TIME_RANGE, srch_bin, event_bin,
    event_stat);
for (i = 0; i < STAT_COUNT; i++) {
    if (event_stat[i] > stat_list[i][STAT_SAVE-1].stat) {
        stat_list[i][STAT_SAVE-1].time = spctgm->start +
            (TRNS_SPECT_WINDOW * SRCH_TIME_RANGE *
            SRCH_TIME_BIN);
        stat_list[i][STAT_SAVE-1].stat = event_stat[i];
        for (j = STAT_SAVE-2; (j >= 0) & (stat_list[i][j].stat < stat_list[i][j+1].stat);
            j--) {
            SWAP(stat_list[i][j].time, stat_list[i][j+1].time, swap.time);
            SWAP(stat_list[i][j].stat, stat_list[i][j+1].stat, swap.stat);
            if (j == STAT_SPCTGM_SAVE - 1) {
                spctgm.filename = sprintfalloc("trns_stat-%i_%s-%i_spctgm-%02.0f",
                    i, filename_id, channel, stat_list[i][j].ord);
                spctgmWrite(spctgm_filename, spctgm, spctgm_sampling_rate,
                    TRNS_SPECT_WINDOW * SRCH_TIME_LENGTH, "Sampling:
                    %f\nWindow: %f\nStart: %f\nStep: %f\nSize: %i
                    %i\nEvent: %i %i %i %i\nStatistic: %f\nOrdinal: %i\n",
                    sampling_rate, TRNS_SPECT_WINDOW *
                    TRNS_SPECT_OVERLAP, spctgm->start,
                    TRNS_SPECT_WINDOW, trnsbuf[channel].SPECT,
                    SRCH_TIME_LENGTH, f * SRCH_FREQ_BIN,
                    SRCH_TIME_RANGE * SRCH_TIME_BIN, SRCH_FREQ_BIN *
                    SRCH_FREQ_OVERLAP, SRCH_TIME_BIN *
                    SRCH_TIME_OVERLAP, stat_list[i][j].stat, stat_list[i][j].ord);
                free(spctgm_filename);
            }
        }
        else {
            SWAP(stat_list[i][j].ord, stat_list[i][j+1].ord, swap.ord);

```

```

    }
    }
    }
    }
    }
    bufferShift(spctgm, spctgm_sampling_rate, spctgm->start +
                TRNS_SPECT_WINDOW * SRCH_TIME_BIN);
    }
    for (i = 0; i < STAT_COUNT; i++) {
        algrWrite(stat_list_filename[i], STAT_SAVE, sizeof(stat_list[i][0]), stat_list[i], "");
    }
    for (i = 0; i < STAT_COUNT; i++) {
        free(stat_list_filename[i]);
    }
    return;
}

```

A.1.7 Miscellaneous subroutines and functions

Listing A.24: The algrCross subroutine

```

void algrCross(int N, fftw_real *ft1, fftw_real *ft2, fftw_real *cross, fftw_real norm, int
              add) {
    int i;
    if (!add) {
        for (i = 0; i < N+2; i++) {
            cross[i] = 0.0;
        }
    }
    cross[0] += ft1[0]*ft2[0] / norm;
    for (i = 1; i < (N+1)/2; i++) {
        cross[i] += (ft1[i]*ft2[i] + ft1[N-i]*ft2[N-i]) / norm;
        cross[N/2+1+i] += (ft1[i]*ft2[N-i] - ft2[i]*ft1[N-i]) / norm;
    }
    if (N % 2 == 0) {
        cross[N/2] += ft1[N/2]*ft2[N/2] / norm;
    }
    return;
}

```

Listing A.25: The algrCrossSqrD subroutine

```

void algrCrossSqrD(int N, fftw_real *ft1, fftw_real *ft2, fftw_real *cross2, fftw_real norm,
                  int add) {
    int i;
    fftw_real norm2;
    if (!add) {
        for (i = 0; i < N+2; i++) {
            cross2[i] = 0.0;
        }
    }
}

```

```

}
norm2 = pow(norm, 2);
cross2[0] += pow(ft1[0]*ft2[0], 2) / norm2;
for (i = 1; i < (N+1)/2; i++) {
    cross2[i] += (pow(ft1[i]*ft2[i] + ft1[N-i]*ft2[N-i], 2) + pow(ft1[i]*ft2[N-i] -
        ft2[i]*ft1[N-i], 2)) / norm2;
}
if (N % 2 == 0) {
    cross2[N/2] += pow(ft1[N/2]*ft2[N/2], 2) / norm2;
}
return;
}

```

Listing A.26: The algrRead function

```

int algrRead(char *filename, int N, int size, void *data, char *format, ...) {
    char *data_filename, *log_filename;
    FILE *fdata, *flog;
    va_list ap;
    int init;
    data_filename = sprintfalloc("%s.dat", filename);
    log_filename = sprintfalloc("%s.log", filename);
    fdata = NULL;
    fdata = fopen(data_filename, "rb");
    init = (fdata == NULL);
    if (init) {
        flog = fopen(log_filename, "w");
        va_start(ap, format);
        fprintf(flog, format, ap);
        va_end(ap);
        fclose(flog);
    }
    else {
        fread(data, size, N, fdata);
        fclose(fdata);
    }
    free(data_filename);
    free(log_filename);
    return init;
}

```

Listing A.27: The algrWrite subroutine

```

void algrWrite(char *filename, int N, int size, void *data, char *format, ...) {
    char *data_filename, *log_filename;
    FILE *fdata, *flog;
    va_list ap;
    data_filename = sprintfalloc("%s.dat", filename);
    log_filename = sprintfalloc("%s.log", filename);
    fdata = fopen(data_filename, "wb");

```

```

fwrite(data, size, N, fdata);
fclose(fdata);
flog = fopen(log_filename, "a");
va_start(ap, format);
vfprintf(flog, format, ap);
va_end(ap);
fclose(flog);
free(data_filename);
free(log_filename);
return;
}

```

Listing A.28: The bufferAdd subroutine

```

void bufferAdd(Buffer *buffer, double buffer_sampling_rate, fftw_real *data, double
    data_start, double data_length, double data_sampling_rate) {
    int i, j;
    double r;
    if (buffer->init == 0) {
        buffer->start = data_start;
        buffer->init = 1;
    }
    if (data_start + data_length > buffer->start) {
        if (data_start + data_length > buffer->start + buffer->length) {
            i = (int) floor(buffer->length * buffer_sampling_rate);
            buffer->length = data_start + data_length - buffer->start;
            buffer->buffer = (fftw_real*) realloc(buffer->buffer, (int) floor(buffer->length *
                buffer_sampling_rate) * sizeof(fftw_real));
            for (; i < (int) floor(buffer->length * buffer_sampling_rate); i++) {
                buffer->buffer[i] = 0;
            }
        }
        j = (int) floor((data_start - buffer->start) * buffer_sampling_rate);
        if (j < 0) {
            i = -j;
        }
        else {
            i = 0;
        }
        if (data_sampling_rate == buffer_sampling_rate) {
            for (; i < data_length * buffer_sampling_rate; i++) {
                buffer->buffer[j + i] += data[i];
            }
        }
        else {
            r = data_sampling_rate / buffer_sampling_rate;
            for (; i < (int) floor(data_length * buffer_sampling_rate); i++) {
                buffer->buffer[j + i] += data[(int) floor(i * r)];
            }
        }
    }
}

```

```

    }
}
return;
}

```

Listing A.29: The `bufferFree` subroutine

```

void bufferFree(Buffer *buffer) {
    if (buffer->buffer != NULL) {
        free(buffer->buffer);
    }
    bufferInit(buffer);
    return;
}

```

Listing A.30: The `bufferInit` subroutine

```

void bufferInit(Buffer *buffer) {
    buffer->buffer = NULL;
    buffer->start = 0;
    buffer->init = 0;
    buffer->length = 0;
}

```

Listing A.31: The `bufferShift` subroutine

```

void bufferShift(Buffer *buffer, double sampling_rate, double to_start) {
    if (to_start > buffer->start) {
        buffer->length -= (to_start - buffer->start);
        if (buffer->length > 0) {
            buffer->buffer = (fftw_real*) memmove(&buffer->buffer[0],
                &buffer->buffer[(int) floor((to_start - buffer->start) * sampling_rate)],
                (int) floor(buffer->length * sampling_rate) * sizeof(fftw_real));
            buffer->buffer = (fftw_real*) realloc(buffer->buffer, (int) floor(buffer->length *
                sampling_rate) * sizeof(fftw_real));
        }
        else {
            buffer->length = 0;
            free(buffer->buffer);
            buffer->buffer = NULL;
        }
        buffer->start = to_start;
    }
    return;
}

```

Listing A.32: The `error` subroutine

```

int error(int ifrank, int eqrank, char *format, ...) {
    char *str;
    va_list ap;

```



```

if (ifrank == eqrank) {
    va_start(ap, format);
    fprintf(stderr, "ERROR: ");
    vfprintf(stderr, format, ap);
    fprintf(stderr, "\n");
    va_end(ap);
}
return EXIT_FAILURE;
}

```

Listing A.33: The longFinal subroutine

```

void longFinal(int CHANNELS) {
    int i;
    for (i = 0; i < CHANNELS; i++) {
        free(longbuf[i].fft_in);
        free(longbuf[i].fft_out);
        free(longbuf[i].fft_win);
        free(longbuf[i].spect);
        free(longbuf[i].corr);
        free(longbuf[i].autc);
        rfftw_destroy_plan(longbuf[i].fft_plan);
    }
    free(longbuf);
    return;
}

```

Listing A.34: The longInit subroutine

```

void longInit(int CHANNELS, double sampling_rate[ ]) {
    int i, j;
    longbuf = (LongBuf*) malloc(CHANNELS * sizeof(LongBuf));
    for (i = 0; i < CHANNELS; i++) {
        longbuf[i].FFT = LONG_SPECT_WINDOW * LONG_SPECT_OVERLAP *
            sampling_rate[i];
        longbuf[i].fft_in = (fftw_real*) malloc(2 * longbuf[i].FFT * sizeof(fftw_real));
        longbuf[i].fft_out = (fftw_real*) malloc(2 * longbuf[i].FFT * sizeof(fftw_real));
        longbuf[i].fft_win = (fftw_real*) malloc(longbuf[i].FFT * sizeof(fftw_real));
        longbuf[i].fft_win_norm = 0;
        longbuf[i].SPECT = longbuf[i].FFT / 2 + 1;
        longbuf[i].spect = (fftw_real*) malloc(2 * longbuf[i].SPECT * sizeof(fftw_real));
        longbuf[i].CORR = (2 * LONG_CORR_SHIFT * sampling_rate[i]) + 1;
        longbuf[i].corr = (fftw_real*) malloc(longbuf[i].CORR * sizeof(fftw_real));
        longbuf[i].AUTC = (LONG_CORR_SHIFT * sampling_rate[i]) + 1;
        longbuf[i].autc = (fftw_real*) malloc(longbuf[i].AUTC * sizeof(fftw_real));
        longbuf[i].fft_plan = rfftw_create_plan_specific(longbuf[i].FFT, FFTW_FORWARD,
            FFTW_PLAN_FLAGS, longbuf[i].fft_in, 1, longbuf[i].fft_out, 1);
    }
    for (i = 0; i < CHANNELS; i++) {
        longbuf[i].fft_win_norm = 0.0;
    }
}

```

```

    for (j = 0; j < longbuf[i].FFT; j++) {
        longbuf[i].fft_win[j] = 1 - fabs((fftw_real) (2*j - longbuf[i].FFT) / (fftw_real)
            longbuf[i].FFT);
        longbuf[i].fft_win_norm += longbuf[i].fft_win[j] * longbuf[i].fft_win[j];
    }
    longbuf[i].fft_win_norm *= longbuf[i].FFT;
}
return;
}

```

Listing A.35: The `qsort_double_abs` function

```

int qsort_double_abs(const void *a, const void *b) {
    double s;
    s = abs(*(double*) a) - abs(*(double*) b);
    return (s < 0) ? -1 : ((s > 0) ? 1 : 0);
}

```

Listing A.36: The `qsort_fftw_real` function

```

int qsort_fftw_real(const void *a, const void *b) {
    double s;
    s = *((fftw_real*) a) - *((fftw_real*) b);
    return (s < 0) ? -1 : ((s > 0) ? 1 : 0);
}

```

Listing A.37: The `spctgmWrite` subroutine

```

void spctgmWrite(char *filename, Buffer *spctgm, double sampling_rate, double length,
    char *format, ...) {
    char *data_filename;
    char *log_filename;
    FILE *fdata;
    FILE *flog;
    va_list ap;
    data_filename = sprintfalloc("%s.dat", filename);
    log_filename = sprintfalloc("%s.log", filename);
    fdata = fopen(data_filename, "wb");
    fwrite(spctgm->buffer, sizeof(fftw_real), (int) floor(length * sampling_rate), fdata);
    fclose(fdata);
    flog = fopen(log_filename, "w");
    va_start(ap, format);
    fprintf(flog, format, ap);
    va_end(ap);
    fclose(flog);
    free(data_filename);
    free(log_filename);
    return;
}

```

Listing A.38: The `sprintfalloc` function

```

char *sprintfalloc(char *format, ...) {
    char *str;
    va_list ap;
    va_start(ap, format);
    str = (char*) malloc((vsnprintf(NULL, 0, format, ap) + 1) * sizeof(char));
    va_end(ap);
    va_start(ap, format);
    vsprintf(str, format, ap);
    va_end(ap);
    return str;
}

```

Listing A.39: The `statCompute` subroutine

```

void statCompute(int BIN, int SRCH_BIN, fftw_real **srch_bin, fftw_real *event_bin,
    double *event_stat) {
    int s, i, j;
    double event_bin_mean;
    double event_bin_zero_mean[BIN];
    double srch_bin_mean;
    double diff_mean_sqrd;
    double diff[BIN];
    double diff_sqrd;
    int rank;
    int rank_m;
    double max_diff;
    event_bin_mean = 0;
    for (i = 0; i < BIN; i++) {
        event_bin_mean += event_bin[i];
    }
    event_bin_mean /= BIN;
    for (i = 0; i < BIN; i++) {
        event_bin_zero_mean[i] = event_bin[i] - event_bin_mean;
    }
    event_stat[0] = 0;
    event_stat[1] = 0;
    for (s = 0; s < SRCH_BIN; s++) {
        srch_bin_mean = 0;
        for (i = 0; i < BIN; i++) {
            srch_bin_mean += srch_bin[s][i];
        }
        srch_bin_mean /= BIN;
        diff_sqrd = 0;
        for (i = 0; i < BIN; i++) {
            diff_sqrd += pow(srch_bin[s][i] - srch_bin_mean - event_bin_zero_mean[i], 2);
        }
        diff_sqrd = (diff_sqrd == 0) ? 1 : diff_sqrd;
        diff_mean_sqrd = pow(srch_bin_mean - event_bin_mean, 2);
    }
}

```

```

    event_stat[0] += diff_mean_sqrd;
    event_stat[1] += diff_mean_sqrd / diff_sqrd;
}
event_stat[0] = sqrt(event_stat[0] / SRCH_BIN);
event_stat[1] = sqrt(event_stat[1] / SRCH_BIN);
event_stat[2] = 0;
event_stat[3] = 0;
for (s = 0; s < SRCH_BIN; s++) {
    for (i = 0; i < BIN; i++) {
        diff[i] = srch_bin[s][i] - event_bin[i];
    }
    qsort(diff, BIN, sizeof(diff[0]), qsort_double_abs);
    rank = rank_m = 0;
    for (i = 0; i < BIN; i++) {
        if (diff[i] > 0) {
            rank += i;
        }
        else if (diff[i] < 0) {
            rank_m += i;
        }
    }
    event_stat[2] += pow((rank > rank_m) ? rank : rank_m, 2);
    event_stat[3] += pow(diff[BIN-1], 2);
}
event_stat[2] = sqrt(event_stat[2] / SRCH_BIN);
event_stat[3] = sqrt(event_stat[3] / SRCH_BIN);
event_stat[4] = 0;
qsort(event_bin, BIN, sizeof(event_bin[0]), qsort_fftw_real);
for (s = 0; s < SRCH_BIN; s++) {
    rank = 0;
    for (i = 0; i < BIN; i++) {
        for (j = 0; (j < BIN) & (event_bin[j] < srch_bin[s][i]); j++, rank++);
    }
    event_stat[4] += pow(rank - BIN*(BIN-1)/2, 2);
}
event_stat[4] = sqrt(event_stat[4] / SRCH_BIN);
return;
}

```

Listing A.40: The strcatfalloc subroutine

```

void strcatfalloc(char **s, char *format, ...) {
    char *str;
    va_list ap;
    va_start(ap, format);
    str = (char*) malloc((vsnprintf(NULL, 0, format, ap) + 1) * sizeof(char));
    va_end(ap);
    va_start(ap, format);
    vsprintf(str, format, ap);
}

```

```

va_end(ap);
if (*s == NULL) {
    *s = str;
}
else {
    *s = (char*) realloc(*s, (strlen(*s) + strlen(str) + 1) * sizeof(char));
    strcat(*s, str);
    free(st);
}
return;
}

```

Listing A.41: The trnsFinal subroutine

```

void trnsFinal(int CHANNELS) {
    int i;
    for (i = 0; i < CHANNELS; i++) {
        free(trnsbuf[i].fft_in);
        free(trnsbuf[i].fft_out);
        free(trnsbuf[i].fft_win);
        free(trnsbuf[i].spect);
        rfftw_destroy_plan(trnsbuf[i].fft_plan);
    }
    free(trnsbuf);
    free(event_bin);
    for (i = 0; i < (2 * SRCH_TIME_RANGE); i++) {
        free(srch_bin[i]);
    }
    free(srch_bin);
    return;
}

```

Listing A.42: The trnsInIt subroutine

```

void trnsInIt(int CHANNELS, double sampling_rate[ ]) {
    int i, j;
    trnsbuf = (TrnsBuf*) malloc(CHANNELS * sizeof(TrnsBuf));
    for (i = 0; i < CHANNELS; i++) {
        trnsbuf[i].FFT = TRNS_SPECT_WINDOW * TRNS_SPECT_OVERLAP *
            sampling_rate[i];
        trnsbuf[i].fft_in = (fftw_real*) malloc(2 * trnsbuf[i].FFT * sizeof(fftw_real));
        trnsbuf[i].fft_out = (fftw_real*) malloc(2 * trnsbuf[i].FFT * sizeof(fftw_real));
        trnsbuf[i].fft_win = (fftw_real*) malloc(trnsbuf[i].FFT * sizeof(fftw_real));
        trnsbuf[i].fft_win_norm = 0;
        trnsbuf[i].SPECT = trnsbuf[i].FFT/2 + 1;
        trnsbuf[i].spect = (fftw_real*) malloc(2 * trnsbuf[i].SPECT * sizeof(fftw_real));
        trnsbuf[i].fft_plan = rfftw_create_plan_specific(trnsbuf[i].FFT, FFTW_FORWARD,
            FFTW_PLAN_FLAGS, trnsbuf[i].fft_in, 1, trnsbuf[i].fft_out, 1);
    }
    for (i = 0; i < CHANNELS; i++) {

```

```

trnsbuf[i].fft_win_norm = 0.0;
for (j = 0; j < trnsbuf[i].FFT; j++) {
    trnsbuf[i].fft_win[j] = 1 - fabs((fftw_real) (2*j - trnsbuf[i].FFT) / (fftw_real)
        trnsbuf[i].FFT);
    trnsbuf[i].fft_win_norm += trnsbuf[i].fft_win[j] * trnsbuf[i].fft_win[j];
}
trnsbuf[i].fft_win_norm *= trnsbuf[i].FFT;
}
event_bin = (fftw_real*) malloc(SRCH_TIME_BIN * SRCH_TIME_OVERLAP *
    SRCH_FREQ_BIN * SRCH_FREQ_OVERLAP * sizeof(fftw_real));
srch_bin = (fftw_real**) malloc((2 * SRCH_TIME_RANGE) * sizeof(fftw_real*));
for (i = 0; i < (2 * SRCH_TIME_RANGE); i++) {
    srch_bin[i] = (fftw_real*) malloc(SRCH_TIME_BIN * SRCH_TIME_OVERLAP *
        SRCH_FREQ_BIN * SRCH_FREQ_OVERLAP * sizeof(fftw_real));
}
return;
}

```

A.2 The MATLAB post-processing code

Listing A.43: The MATLAB post-processing code

```

function process(varargin)
global args
args = varargin;
channel = {'0', '1', '2', '3'};
channeltitle = {'horizontal seismic', 'vertical seismic', 'mains voltage',
    'magnetic'};
numchannels = length(channel);
f = false;
for k = 1:numchannels
    f = f || strcmp(['k', channel{k}]);
end
if ~f
    args = {'k', args{:}};
end
stat = {'0', '1', '2', '3', '4'};
stattitle = {'Difference-mean', 'Student t', 'Wilcoxon signed rank',
    'Kolmogorov-Smirnov', 'Wilcoxon-Mann-Whitney'};
numstats = length(stat);
numspctgm = 20;
if strcmp('x')
    close all;
end
for d = dir('.')
    if d.isdir && strcmp(d.name)
        directory = ['. \', d.name, '\'];
        name = [directory, d.name, '_'];
        load([directory, 'processdata.mat'], 'datatitle', 'datafrom', 'datato',

```

```

        'sitechoose');
datefrom = datenum([1980, 1, 6, 0, 0, datafrom - 13]);
dateto = datenum([1980, 1, 6, 0, 0, datato - 13]);
datatitle = sprintf('\\%s : UTC %s %s to %s %s\n', datatitle, ...
    datestr(datefrom, 'dd/mm/yy'), datestr(datefrom, 'HH:MM:SS'), ...
    datestr(dateto, 'dd/mm/yy'), datestr(dateto, 'HH:MM:SS'));
site = {'aciga', 'hanford', 'livingston', 'virgo'};
sitetitle = {'ANU', 'LHO', 'LLO', 'VIRGO'};
site = {site{sitechoose}};
sitetitle = {sitetitle{sitechoose}};
numsites = length(site);
if argcmp('a')
    for k = 1:numchannels
        if argcmp('k') || argcmp(['k', channel{k}])
            for i = 1:numsites
                file = sprintf('long_auto_%s-%s', site{i}, channel{k});
                namedfigure('%slong_auto_%s-%s', name, site{i}, channel{k});
                f = fopen([directory, file, '.log']);
                sampl = fscanf(f, 'Sampling: %f\n');
                shift = fscanf(f, 'hifit: %f\n');
                fclose(f);
                f = fopen([directory, file, '.dat']);
                auto = fread(f, inf, 'double');
                auto = [auto(end:-1:2); auto];
                fclose(f);
                time = linspace(-shift, shift, length(auto));
                subaxes(1, 1, 1, 1);
                plot(time, auto);
                axis tight;
                title(sprintf('%sAutocorrelation of %s %s', datatitle, sitetitle{i},
                    channeltitle{k}));
                xlabel(sprintf('Time shift of %s relative to itself / s',
                    sitetitle{i}));
                printfigure;
                clear time auto;
            end
        end
    end
end
if argcmp('c')
    for k = 1:numchannels
        if argcmp('k') || argcmp(['k', channel{k}])
            for i = 1:numsites
                for j = (i+1):numsites
                    file = sprintf('long_corr_%s-%s-%s', site{i}, site{j}, channel{k});
                    namedfigure('%slong_corr_%s-%s-%s', name, site{i}, site{j},
                        channel{k});
                    f = fopen([directory, file, '.log']);

```

```

    sampl = fscanf(f, 'Sampling: %f\n');
    shift = fscanf(f, 'hifft: %f\n');
    fclose(f);
    f = fopen([directory, file, '.dat']);
    corr = fread(f, inf, 'double');
    fclose(f);
    time = linspace(-shift, shift, length(corr));
    subaxes(1, 1, 1, 1);
    plot(time, corr);
    axis tight;
    title(sprintf('%sCorrelation of %s-%s %s', datatitle, sitetitle{i},
                 sitetitle{j}, channeltitle{k}));
    xlabel(sprintf('Time shift of %s relative to %s / s', sitetitle{i},
                 sitetitle{j}));
    printfigure;
    clear time corr;
end
end
end
end
if argcmp('p')
    for k = 1:numchannels
        if argcmp('k') || argcmp(['k', channel{k}])
            for i = 1:numsites
                file = sprintf('long_power_%s-%s', site{i}, channel{k});
                namedfigure('%slong_power_%s-%s', name, site{i}, channel{k});
                f = fopen([directory, file, '.log'], 'r');
                sampl = fscanf(f, 'Sampling: %f\n');
                len = fscanf(f, 'Length: %f\n');
                fclose(f);
                f = fopen([directory, file, '.dat'], 'r');
                power = fread(f, inf, 'double');
                fclose(f);
                freq = linspace(0, sampl/2, length(power));
                subaxes(1, 1, 1, 1);
                semilogy(freq, power / max(power));
                axis tight;
                title(sprintf('%sNormalised power spectrum of %s %s', datatitle,
                             sitetitle{i}, channeltitle{k}));
                xlabel(sprintf('Frequency / Hz (Resolution = %0.0g)', 1 / len));
                printfigure;
                clear freq power;
            end
        end
    end
end
end
if argcmp('h')

```



```

for k = 1:numchannels
  if argcmp('k') || argcmp(['k', channel{k}])
    for i = 1:numsites
      for j = (i+1):numsites
        file = sprintf('long_cross_%s-%s-%s', site{i}, site{j}, channel{k});
        files{1} = sprintf('long_power_%s-%s', site{i}, channel{k});
        files{2} = sprintf('long_power_%s-%s', site{j}, channel{k});
        namedfigure('%slong_coher_%s-%s-%s', name, site{i}, site{j},
                    channel{k});
        f = fopen([directory, file, '.log'], 'r');
        sampl = fscanf(f, 'Sampling: %f\n');
        len = fscanf(f, 'Length: %f\n');
        f = fopen([directory, file, '.dat'], 'r');
        coher = fread(f, inf, 'double');
        coher = coher(1:(length(coher)/2)).^2 +
                coher(((length(coher)/2)+1):end).^2;
        fclose(f);
        for n = 1:length(files)
          f = fopen([directory, files{n}, '.dat'], 'r');
          coher = coher ./ fread(f, inf, 'double');
          fclose(f);
        end
        freq = linspace(0, sampl/2, length(coher));
        subaxes(1, 1, 1, 1);
        plot(freq, coher);
        axis tight;
        title(sprintf('Coherence of %s-%s %s', datatitle, sitetitle{i},
                    sitetitle{j}, channeltitle{k}));
        xlabel(sprintf('Frequency / Hz (Resolution = %0.0g)', 1 / len));
        printfigure;
        clear freq coher;
      end
    end
  end
end
for s = 1:numstats
  if argcmp('t') || argcmp(['t', stat{s}])
    for k = 1:numchannels
      if argcmp('k') || argcmp(['k', channel{k}])
        for i = 1:numsites
          for j = (i+1):numsites
            file = sprintf('trns_stat-%s-%s-%s-%s', stat{s}, site{i}, site{j},
                        channel{k});
            namedfigure('%strns_stat-%s-%s-%s-%s', name, stat{s}, site{i},
                        site{j}, channel{k})
            f = fopen([directory, file, '.dat'], 'r');
            data = fread(f, [3, inf], 'double');
          end
        end
      end
    end
  end

```

```

times{i,j} = data(1,:);
statistics{i,j} = data(2,:);
ordinals{i,j} = data(3,:);
clear data;
fclose(f);
subaxes(1, 1, 1, 1);
title(sprintf('%%s%%s statistic for %%s-%%s %%s', datatitle, stattitle{s},
...
    sitetitle{i}, sitetitle{j}, channeltitle{k}));
axis off;
subaxes(1, 2, 1, 1);
plot(times{i,j}, statistics{i,j}, '.');
axis tight;
dates(gca, 'X', 86400, 1, 3, 'dd/mm/yy');
xlabel('Time');
ylabel(sprintf('%%s statistic', stattitle{s}));
subaxes(1, 2, 1, 2);
semilogy(1:length(statistics{i,j}), statistics{i,j});
axis tight;
xlabel('Rank');
ylabel(sprintf('%%s statistic', stattitle{s}));
printfigure;
if argcmp('s')
    for n = 1:numspctgm
        file = sprintf('trns_stat-%%s_%%s-%%s-%%s_spctgm-%%02.Of',
            stat{s}, ...
            site{i}, site{j}, channel{k}, ordinals{i, j}(n));
        namedfigure('%strns_stat-%%s_%%s-%%s-%%s_event-%%02.Of', name,
            stat{s}, site{i}, site{j}, channel{k}, n)
        f = fopen([directory, file, '.log'], 'r');
        sampl = fscanf(f, 'Sampling: %f\n');
        window = fscanf(f, 'Window: %f\n');
        start = fscanf(f, 'Start: %f\n');
        step = fscanf(f, 'ep: %f\n');
        siz = fscanf(f, 'Size: %f %f\n');
        event = fscanf(f, 'Event: %f %f %f %f\n');
        statistic = fscanf(f, 'Statistic: %f\n');
        ordinal = fscanf(f, 'Ordinal: %f\n');
        fclose(f);
        time = start + step*event(2);
        f = fopen([directory, file, '.dat'], 'r');
        spctgm = fread(f, siz, 'double');
        fclose(f);
        for b = 1:size(spctgm,2)
            spctgm(:,b) = log(spctgm(:,b) - min(spctgm(:,b)) + 1);
        end
        spctgmx = linspace(0, sampl/2 + 1, siz(1));
        spctgmy = start + (0:step:(step*(siz(2)-1)));

```

```

eventx =
    [event(1),event(1)+event(3),event(1)+event(3),event(1),event(1)];
eventx = (eventx / siz(1) * (max(spctgmx) - min(spctgmx))) +
    min(spctgmx);
eventy =
    [event(2),event(2),event(2)+event(4),event(2)+event(4),event(2)];
eventy = (eventy / siz(2) * (max(spctgmy) - min(spctgmy))) +
    min(spctgmy);
subaxes(1, 1, 1, 1);
title(sprintf('%%s statistic for %s-%s %s event #%i',
    datatitle, stattitle{s}, ...
    sitetitle{i}, sitetitle{j}, channeltitle{k}, n));
axis off;
subaxes(1, 2, 1, 1);
plot(times{i,j}, statistics{i,j}, '. ', time, statistic, 'o');
axis tight;
dates(gca, 'X', 86400, 1, 3, 'dd/mm/yy');
xlabel('UTC time');
ylabel(sprintf('%s statistic', stattitle{s}));
subaxes(1, 2, 1, 2);
colormap(jet(256));
contourf(spctgmx, spctgmy, spctgm);
shading flat; hold on;
plot(eventx, eventy, 'k-');
set(gca, 'XLim', [min(spctgmx), max(spctgmx)]);
set(gca, 'YLim', [min(spctgmy), max(spctgmy)]);
dates(gca, 'Y', 1, 1, 3, 'HH:MM:SS');
xlabel('Frequency / Hz');
ylabel(['UTC time from ', datestr(datenum([1980, 1, 6, 0, 0,
    min(spctgmy) - 13]), 'dd/mm/yy')]);
printfigure;
clear spctgm spctgmx spctgmy;
end
end
end
end
end
end
end
end
end
end
function f = argcmp(str)
global args
f = false; v = 1;
while (~f & v <= length(args))
    f = strcmpi(args{v}, str);
    v = v + 1;

```

```

end
function namedfigure(varargin)
set(figure, 'Name', sprintf(varargin{:}), 'NumberTitle', 'off');
set(gcf, 'PaperType', 'A4', 'PaperOrientation', 'portrait');
set(gcf, 'PaperUnits', 'normalized', 'PaperPosition', [0, 0, 0.5, 0.25]);
function subaxes(varargin)
rows = varargin{1}; cols = varargin{2};
r = varargin{3}; c = varargin{4};
left = 0.14; bottom = 0.14; right = 0.02; top = 0.14;
pos = [(c - 1)/cols, (rows - r)/rows, 1/cols, 1/rows] + [left, bottom, -(right+left,
top+bottom)];
axes('Position', pos);
function printfigure
for j = reduce(get(gcf, 'Children'))
if strcmpi(get(j, 'Type'), 'axes')
set(j, 'XTickMode', 'manual');
set(j, 'YTickMode', 'manual');
end
end
if argcmp('j')
print(gcf, '-djpeg', get(gcf, 'Name'));
end
if argcmp('e')
print(gcf, '-depsc', get(gcf, 'Name'));
end
if argcmp('x1')
close(gcf)
end
pack;
function b = reduce(a);
if iscell(a)
b = [];
for i = 1:prod(size(a))
b = [b, reduce(a{1})];
end
else
b = a(:)';
end
function dates(axis, xyz, unit, incr, every, format)
range = get(axis, [xyz, 'Lim']);
time = (ceil(range(1) / unit) * unit) + 13;
while time > range(1)
time = time - unit;
end
tick = []; ticklabel = [];
count = 0;
while time <= range(2)
date = [1980, 1, 6, 0, 0, time - 13];

```

```
tick = [tick; time];
if mod(count, every) == 0
    ticklabel = [ticklabel; datestr(datetime(date), format)];
else
    ticklabel = [ticklabel; blanks(size(ticklabel, 2))];
end
time = time + unit*incr;
count = count + 1;
end
set(gca, [xyz, 'Tick'], tick, [xyz, 'TickLabel'], ticklabel, 'Layer', 'top');
```

Bibliography

- [1] G. Holton, *Thematic Origins of Scientific Thought: Kepler to Einstein*, Harvard University Press, 1973.
- [2] C. M. Will, *Theory and experiment in gravitational physics*, Cambridge University Press, 1981.
- [3] P. R. Saulson, *Fundamentals of Interferometric Gravitational Wave Detectors*, World Scientific, 1994.
- [4] B. F. Schutz, ‘Gravitational Wave Astronomy’, *Classical Quantum Gravity* **16**, A131–A156 (1999).
- [5] The Laser Interferometer Gravitational Wave Observatory, <http://www.ligo.caltech.edu>.
- [6] The VIRGO Project, <http://www.virgo.infn.it/>.
- [7] The GEO600 Project, <http://www.geo600.uni-hannover.de/>.
- [8] The TAMA300 Project, <http://tamago.mtk.nao.ac.jp/>.
- [9] D. Sigg et al., ‘Characterization Of Environmental And Input Beam Noise Inputs’, Technical note LIGO-G000115-00-D, LIGO, 2000, <http://www.ligo.caltech.edu/docs/G/G000115-00.pdf>.
- [10] M. Fyffe, J. Kovalik, D. Lormand, S. Marka, P. Saulson, R. Wooley, J. Romano, R. Luna, M. Casquette, and A. Zermeno, ‘PEM Audit at LIGO Livingston’, Technical note LIGO-G000258-00-D, LIGO, 2000, <http://www.ligo.caltech.edu/docs/G/G000258-00.pdf>.
- [11] R. Schofield, M. Ito, R. Rahkola, E. Mauceli, R. Frey, D. Strom, and J. Brau, ‘Some Effects of Earthquakes, Temperature, Wind Storms and Barometric Pressure on the Interferometer at Hanford’, Technical note LIGO-G000088-00-D, LIGO, 2000, <http://www.ligo.caltech.edu/docs/G/G000088-00.pdf>.
- [12] R. Schofield, S. Mukherjee, R. Rahkola, and J. Sylvestre, ‘Environmental Disturbances: E5, E6 and E7 Investigations’, Technical note LIGO-G020252-00-Z, LIGO, 2002, <http://www.ligo.caltech.edu/docs/G/G020252-00.pdf>.
- [13] N. Christensen, ‘E4 Correlations: Detector Charactersation’, Technical note LIGO-G010315-00-Z, LIGO, 2001, <http://www.ligo.caltech.edu/docs/G/G010315-00.pdf>.
- [14] R. Schofield, G. Gonzalez, M. Landry, and P. Sutton, ‘Intersite Environmental Transients: E5, E6, and E7 investigations’, Technical note LIGO-G020253-00-Z, LIGO, 2002, <http://www.ligo.caltech.edu/docs/G/G020253-00.pdf>.

- [15] R. Schofield, ‘S2 Intersite Environmental Transients Investigation’, Technical note LIGO-G030641-00-Z, LIGO, 2003, <http://www.ligo.caltech.edu/docs/G/G030641-00.pdf>.
- [16] R. Schofield, P. Saulson, and E. Daw, ‘The LIGO E2 Investigation of Non-Stationary Noise’, Technical note LIGO-G010159-00-Z, LIGO, 2001, <http://www.ligo.caltech.edu/docs/G/G010159-00.pdf>.
- [17] R. Schofield, M. Ito, E. Mauceli, H. Radkins, C. Gray, G. Moreno, and G. Gonzalez, ‘Source and Propagation of the Predominant 1-50 Hz Seismic Signal From Off-Site at LIGO-Hanford’, Technical note LIGO-G000262-00-D, LIGO, 2000, <http://www.ligo.caltech.edu/docs/G/G000262-00.pdf>.
- [18] J. Giaime and E. Daw, ‘LLO environmental excitation update’, Technical note LIGO-G030228-00-D, LIGO, 2003, <http://www.ligo.caltech.edu/docs/G/G030228-00.pdf>.
- [19] R. Schofield, E. D’Ambrosio, D. Cook, R. Drever, V. Sannibale, and B. Bland, ‘Environmental Disturbances (Including S1 - Stoppers)’, Technical note LIGO-G020396-00-Z, LIGO, 2002, <http://www.ligo.caltech.edu/docs/G/G020396-00.pdf>.
- [20] R. Schofield, ‘Seismic Measurements at the Stateline Wind Project: And A Prediction of the Seismic Signal that the Proposed Maiden Wind Project Would Produce at LIGO’, Technical note LIGO-T020104-00-Z, LIGO, 2002, <http://www.ligo.caltech.edu/docs/T/T020104-00.pdf>.
- [21] R. Schofield, ‘Progress on S1 Intersite Transients Study’, Technical note LIGO-G030330-00-Z, LIGO, 2003, <http://www.ligo.caltech.edu/docs/G/G030330-00.pdf>.
- [22] R. Schofield, M. Ito, S. Mohanty, S. Penn, R. Rahkola, P. Saulson, and J. Sylvestre, Technical report.
- [23] R. Schofield, A. Ageyev, M. Ito, and B. OReiley, ‘Measurements of Environmental Coupling to the Gravitational Wave Channel (PEM Injections)’, Technical note LIGO-G030297-00-Z, LIGO, 2003, <http://www.ligo.caltech.edu/docs/G/G030297-00.pdf>.
- [24] R. Schofield, J. Hester, N. Christensen, P. Fritschel, M. Ito, S. Klimenko, M. Landry, S. Marka, S. Mohanty, A. Ottewill, and R. Rahkola, ‘Intersite Environmental Correlations: E3 and E4 Investigations. Part 2: Bursts’, Technical note LIGO-G010396-00-Z, LIGO, 2001, <http://www.ligo.caltech.edu/docs/G/G010396-00.pdf>.
- [25] M. Landry, N. Christensen, P. Fritschel, M. Ito, S. Klimenko, S. Marka, S. Mohanty, A. Ottewill, R. Rahkola, R. Schofield, and J. Sylvestre, ‘E3/E4 PEM Correlations, Part I’, Technical note LIGO-G010286-00-H, LIGO, 2001, <http://www.ligo.caltech.edu/docs/G/G010286-00.pdf>.
- [26] N. Christensen, ‘S2 and S3 Interchannel Correlations’, Technical note LIGO-G040148-00-Z, LIGO, 2004, <http://www.ligo.caltech.edu/docs/G/G040148-00.pdf>.

-
- [27] A. Lazzarini, R. Schofield, and A. Vicere, ‘60 Hz Mains Correlations for the U. S. Power Grids’, Technical note LIGO-G020245-00-E, LIGO, 2002, <http://www.ligo.caltech.edu/docs/G/G020245-00.pdf>.
- [28] A. Lazzarini, A. Vicerè, and R. Schofield, ‘Analysis of the effects of long term correlations over long baselines over narrowband features in cross-correlation measurements’, Technical note LIGO-T010101-01-E, LIGO, 2001, <http://www.ligo.caltech.edu/docs/T/T010101-01.pdf>.
- [29] B. Cusack, A. Searle, S. Scott, and D. McClelland, ‘Global Second and Third Order Correlations in Physical Environment Monitors’, Technical note LIGO-G030364-00-Z, LIGO, 2003, <http://www.ligo.caltech.edu/docs/G/G030364-00.pdf>.
- [30] The SETI Institute, <http://www.seti-inst.edu/>.
- [31] Stefan Mayer Instruments, <http://www.stefan-mayer.com/>.
- [32] National Instruments Corporation, <http://www.ni.com/>.
- [33] Symmetricom Inc., <http://www.symmttm.com/>.
- [34] Stanford Research Systems Inc., <http://www.thinkSRS.com/>.
- [35] Apple Computer Inc., <http://www.apple.com/>.
- [36] The ANU Supercomputer Facility, <http://www.anusf.anu.edu.au/>.
- [37] The Mass Data Storage System at the ANU Supercomputer Facility, <http://nf.apac.edu.au/facilities/mdss/>.
- [38] ‘Specification of a Common Data Frame Format for Interferometric Gravitational Wave Detectors’, Technical note LIGO-T970130-F-E/VIRGO-SPE-LAP-5400-102, LIGO, VIRGO, 2002, <http://www.ligo.caltech.edu/docs/T/T970130-F.pdf>.
- [39] B. Mours, *Frame Library User’s Manual*, VIRGO, <http://wwwlapp.in2p3.fr/virgo/FrameL/FrDoc.html>.
- [40] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, second edition, 1988.
- [41] StimpSoft Inc., <http://www.stimpsoft.com/>.
- [42] LIGO Data Analysis System, <http://www.ldas-sw.ligo.caltech.edu/>.
- [43] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, third edition, 1997.
- [44] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computation*, Cambridge University Press, second edition, 1992.
- [45] E. W. Weisstein, *Fourier Transform*, from MathWorld – A Wolfram Web Resource, <http://mathworld.wolfram.com/FourierTransform.html>.
- [46] P. Bloomfield, *Fourier Analysis of Time Series: An Introduction*, John Wiley & Sons, second edition, 2000.

- [47] J. F. James, *A student's guide to Fourier transforms: with applications in physics and engineering*, Cambridge University Press, 1995.
- [48] M. Frigo and S. G. Johnson, *FFTW User's Manual: For version 2.1.5, 16 March 2003*, in <http://www.fftw.org/fftw-2.1.5.tar.gz>.
- [49] G. K. Kanji, *100 Statistical Tests*, SAGE Publications, new edition, 1999.
- [50] E. W. Weisstein, *Paired t-Test*, from MathWorld – A Wolfram Web Resource, <http://mathworld.wolfram.com/Pairedt-Test.html>.
- [51] J. V. Bradley, *Distribution-Free Statistical Tests*, Prentice-Hall, 1968.
- [52] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, second edition, 1999.
- [53] The Message-Passing Interface Forum, <http://www.mpi-forum.org/>.
- [54] APAC National Facility MPI Programming Course, <http://nf.apac.edu.au/training/MPIProg/>.
- [55] LAM/MPI Parallel Computing, <http://www.lam-mpi.org/>.
- [56] The GNU Compiler Collection, <http://gcc.gnu.org/>.
- [57] S. M. Scott, A. C. Searle, B. J. Cusack, A. J. Moylan, D. E. McClelland, D. Coward, R. Burman, E. Howell, and D. Blair, 'ACIGA Data Analysis', Technical note LIGO-G030365-00-Z, Australian National University, University of Western Australia, 2003, <http://www.ligo.caltech.edu/docs/G/G030365-00.pdf>.
- [58] D. Buskulic, I. Fiori, I. Ferrante, F. Marion, and B. Mours, *The Frame Vector Library*, VIRGO, <http://wwwlapp.in2p3.fr/virgo/FrameL/Frv.html>.
- [59] E. J. Daw, J. A. Giaime, D. Lormand, M. Lubinski, and J. Zweizig, 'Long term study of the seismic environment at LIGO', *Classical Quantum Gravity* **21**(9), 2255–2273 (2004).