

**Data Analysis Infrastructure
for
Gravitational Wave Astronomy**

By
Antony Charles Searle

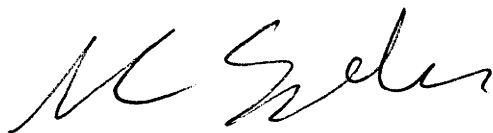
December 2, 2004



*A thesis submitted for the degree of Doctor of Philosophy
of The Australian National University*

Declaration

I certify that the work contained in this thesis is my own original research, produced in collaboration with my supervisor—Dr Susan M. Scott. All material taken from other references is explicitly acknowledged as such. I also certify that the work contained in this thesis has not been submitted for any other degree.

A handwritten signature in black ink, appearing to read 'A. Searle', written in a cursive style.

Antony Charles Searle

Acknowledgements

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of light, it was the season of darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to Heaven, we were all going direct the other way—in short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only.

— Charles Dickens, *A Tale of Two Cities*.

These last few years have been the worst of my life: watching my mother Christine slowly consumed by cancer, my brush with history in Washington on a sunny Tuesday in September, and the very personal war on terror that followed. So special thanks is due to the people who also made these years the best of times:¹

Susan Scott ('for God's sake, don't say *that!*'), my indefatigable one-woman supervisor, funding body, travel agent, motivator, defender, coach and friend.

David McClelland, for his supervision, and for keeping the whole antipodean gravitational-wave show on the road.

Those who shared my offices and kept the air crackling with 'electricity': Mike Ashley (and the Bone of Contention, and Other Tales), Benedict Cusack ('do you think it's an ostrich farm?'), Ingrid Irmer (into every generation a Computer Slayer is born), and Andrew Moylan ('I can't work out what this thing on my forehead is'). They left 'Surly' Searle less so than they found him.

¹All quotes severely out of context.

Those at my LIGO homes-away-from-home: at Caltech, Albert Lazzarini ('American-Indian and Indian-American don't commute. They're, like, *iñ*.'), Kent Blackburn ('I might engage in a little biological warfare of my own'), Ed Maros (who insisted the fault lay with Tcl), Phil Ehrens (who insisted the fault lay with C++), Isaac Salzman (who found the fault), the faultless Peter Shawhan, and the whole LDAS team; at PennState, Sam Finn ('when I get home at night I need a dog to kick, and [name withheld] can be that dog') and Patrick Sutton (who drove all day to collect me after September 11, then nearly hit an eighteen-wheeler on the way back); at UT-Brownsville, Joe Romano (simply the nicest guy... 'nuff said), Warren Anderson ('you can't be *that* gullible'), and John Whelan (who has said so many things so quickly that I only remember the gist of them).

Philip Charlton, Australia's 'mole' in LIGO data analysis, fits most of the above categories, and is a one-man cultural beachhead.

Anna Weatherly ('can you say that again, slowly?') for her counsel, Dr Alison McIntyre for an efficacious pharmacopoeia, and Coca-Cola Amatil for putting life's necessities in a contour bottle.

My friends at the Canberra Speculative Fiction Guild, in particular the asynchronous Maxine MacArthur, Michael Barry (wink-wink, nod-nod, say-no-more), the punctual Alan Price, and the lycanthromorphilic (?) Robbie Matthews. I'm not sure if they reciprocate: 'The CSFG would like to acknowledge the efforts of the following people... Alan [sic] Searle... for additional proofing.'—*Elsewhere* (ed. Michael Barry), CSFG Publishing.

I don't know Fred Raab very well, but this overheard meta-quote is too good to pass up: 'So my quote in her thesis is going to be that LIGO looks like a sewage plant.'

Finally, my father Robert for his financial support and airport taxi service in the wee small hours, my brother Damon for his technical support, and both of them for being there with me through these interesting times.

Abstract

Interferometric gravitational wave observatories are coming on-line around the world, sensitive to infinitesimal ripples propagating through space-time itself. Data analysis assumes an unusual importance in gravitational wave astronomy; all predicted gravitational wave signals from plausible astrophysical scenarios will be at the margins of detectability for current instruments, and even as sensitivities improve, the majority of signals will remain in this regime.

The immediate goal of current observatories is to make the first widely-accepted direct detection of gravitational waves; to this end, I have made significant contributions to the data analysis systems of leading observatories, spanning design, implementation, testing, and characterisation of components ranging from basic signal-processing to tailored data conditioning operations. These components have been employed to produce several worlds-best direct observational upper limits on gravitational wave phenomena.

In the longer term, as gravitational wave astronomy becomes a reality, issues of how to best combine and expand the global network of observatories will come to the fore. I have constructed a suite of models to explore optimal configurations of a global network of observatories for the detection of a variety of proposed gravitational wave source populations, placing particular emphasis on the contribution a proposed Australian gravitational wave observatory would make to the global community.

Contents

| | | |
|----------|--|----------|
| 1 | Introduction | 1 |
| 1.1 | Review | 2 |
| 1.2 | Publications | 4 |
| 1.3 | Overview | 6 |
| I | Data Conditioning | 7 |
| 2 | The LDAS data conditioning API | 9 |
| 2.1 | The LIGO Data Analysis System (LDAS) | 10 |
| 2.1.1 | APIs | 11 |
| 2.1.2 | Command language | 11 |
| 2.2 | Design and evolution | 13 |
| 2.3 | Universal Data Type (UDT) | 16 |
| 2.3.1 | Implementation | 16 |
| 2.3.2 | Scalar | 18 |
| 2.3.3 | Sequence | 20 |
| 2.3.4 | Matrix | 22 |
| 2.3.5 | Metadata | 24 |
| 2.4 | Signal processing | 25 |
| 2.4.1 | Mixer | 26 |
| 2.4.2 | LinFilt | 31 |
| 2.4.3 | Resample | 33 |
| 2.5 | Actions | 36 |
| 2.5.1 | Call chain | 37 |
| 2.5.2 | Call chain function | 38 |
| 2.5.3 | mix | 38 |
| 2.5.4 | Simple actions | 40 |

| | | |
|-----------|--|------------|
| 2.6 | Testing | 41 |
| 2.7 | Summary | 45 |
| 3 | Line removal | 47 |
| 3.1 | Motivation | 49 |
| 3.2 | Design | 54 |
| 3.3 | Characterisation | 57 |
| | 3.3.1 Injection | 58 |
| | 3.3.2 Coherence | 71 |
| 3.4 | Stochastic background S1 upper limit | 74 |
| 3.5 | Conclusion | 79 |
| II | Network Simulation | 81 |
| 4 | Network simulation | 83 |
| 4.1 | Geometrical considerations | 84 |
| | 4.1.1 Interferometric gravitational wave detectors | 84 |
| | 4.1.2 Gravitational wave sources | 85 |
| | 4.1.3 Antenna patterns | 85 |
| | 4.1.4 Implementation | 87 |
| | 4.1.5 Existing and proposed detectors | 90 |
| 4.2 | Figures of merit | 90 |
| 4.3 | Summary | 91 |
| 5 | Geographical configuration | 93 |
| 5.1 | Detection of binary inspiral events | 94 |
| | 5.1.1 Waveform and response | 94 |
| | 5.1.2 Analysis strategies | 95 |
| | 5.1.3 Detection rate | 97 |
| | 5.1.4 Implementation | 98 |
| 5.2 | Results | 100 |
| 5.3 | Conclusion | 104 |
| 6 | Continuous-wave sources | 107 |
| 6.1 | Introduction | 107 |
| 6.2 | Methodology | 107 |
| | 6.2.1 Implementation | 109 |

| | | |
|----------|--|------------|
| 6.3 | Detection | 110 |
| 6.4 | Galactic distribution | 112 |
| 6.5 | Conclusion | 115 |
| 7 | Summary and future directions | 117 |
| 7.1 | Data conditioning | 117 |
| 7.2 | Network simulation | 119 |
| 7.3 | Conclusion | 120 |
| A | Line remover implementation | 121 |
| A.1 | Band selection | 121 |
| A.2 | Output-error model | 124 |
| A.3 | Interface | 131 |
| B | Model of galactic population... | 151 |
| | Bibliography | 157 |

List of Figures

| | | |
|-----|---|----|
| 3.1 | Lines in the power spectrum of the 4 km LIGO Hanford Observatory. | 48 |
| 3.2 | LHO (a) 4 km and (b) 2 km interferometers and (c) LLO 4 km interferometer output power spectra (uncalibrated), before (dotted) and after (solid) line removal. | 51 |
| 3.3 | (a) Coherence of, and (b) accumulated coherence of, H1:LSC-AS_Q and H2:LSC-AS_Q before (dotted) and after (solid) line removal, with the accumulated coherence of H1:LSC-AS_Q and L1:LSC-AS_Q (dashed) provided for reference. | 52 |
| 3.4 | (a) LHO and (b) LLO voltage monitor channel (uncalibrated) power spectra. | 53 |
| 3.5 | Power spectrum of H1:LSC-AS_Q and added sinusoids before (dotted) and after (solid) application of line removal to 300 ± 8 Hz with order 8. The power spectrum of the injected sinusoids alone is dashed. The estimation era was GPS 714974400–714975000 and the line removal era was GPS 714975000–714975600. | 60 |
| 3.6 | Power spectrum of H1:LSC-AS_Q and added sinusoids (dashed) before (dotted) and after (solid) application of line removal to 308 ± 8 Hz with order 8. The estimation era was GPS 714974400–714975000 and the line removal era was GPS 714975000–714975600. The line lies on the edge of the removal interval, and is only slightly attenuated. Other frequencies are unaffected. | 62 |
| 3.7 | Power spectrum of H1:LSC-AS_Q and added sinusoids (dashed) before (dotted) and after (solid) application of line removal to 309 ± 8 Hz with order 8. The estimation era was GPS 714974400–714975000 and the line removal era was GPS 714975000–714975600. The line lies outside the removal interval. No frequencies are significantly affected. | 63 |

- 3.8 Power spectrum of H1:LSC-AS_Q before (dotted) and after (solid) application of line removal to 309 ± 8 Hz with order 8. The estimation era was GPS 714974400–714975000 and the line removal era was GPS 714975000–714975600. 64
- 3.9 Power spectrum of H1:LSC-AS_Q before (dotted) and after (solid) application of line removal to 300 ± 64 Hz with order 8. The estimation era was GPS 714974400–714975000 and the line removal era was GPS 714975000–714975600. The line is partially removed; note the noise floor has been perturbed. 66
- 3.10 Power spectrum of H1:LSC-AS_Q before (dotted) and after (solid) application of line removal to 300 ± 64 Hz with order 8. The estimation era was GPS 714974400–714975000 and the line removal era was GPS 714975000–714975600. The algorithm has introduced noise across its band of operation. 67
- 3.11 Power spectrum of H1:LSC-AS_Q before (dotted) and after (solid) application of line removal to 300 ± 8 Hz with order 1. The estimation era was GPS 714974400–714975000 and the line removal era was GPS 714975000–714975600. The line has been removed to the noise floor; there is little evidence of broadening of the signal. 68
- 3.12 Power spectrum of H1:LSC-AS_Q before (dotted) and after (solid) application of line removal to 300 ± 8 Hz with order 128. The estimation era was GPS 714974400–714975000 and the line removal era was GPS 714975000–714975600. Note perturbation of the noise floor throughout the line removal band. 69
- 3.13 Power spectrum of H1:LSC-AS_Q before (dotted) and after (solid) application of line removal to 300 ± 8 Hz with order 8. The estimation era was GPS 714974996–714975000 and the line removal era was GPS 714975000–714975600. Note perturbation of the noise floor throughout the line removal band. 70
- 3.14 Coherence of (a) H1:LSC-AS_Q, (b) H2:LSC-AS_Q and (c) L1:LSC-AS_Q with their respective voltage monitor channels, H0:PEM-LVEA2_V1 and L0:PEM-LVEA_V1, before (dotted) and after (solid) application of the line removal technique described in §3.3. 72

3.15 Power spectra of the prediction for (a) H1:LSC-AS_Q, (b) H2:LSC-AS_Q and (c) L1:LSC-AS_Q (solid). Corresponding power spectra of the channels are provided for reference (dotted). 73

3.16 H1-H2 coherence with (red) and without (blue) the line removal stage of the stochastic pipeline. 75

3.17 Per-data-segment (\times) and total (horizontal lines) upper limit results with and without line removal, showing no significant differences. The dashed lines are 90% confidence bounds on the (solid line) limit. 76

3.18 The power spectrum of the injected line (red) compared to that of the 4 km Hanford interferometer (blue). 77

3.19 Structural comparison of the search’s optimal filter (top) and H1 and H2 spectra (bottom), showing that deep notches in the filter correspond to spectral lines. 78

4.1 Antenna pattern $(F_+^2 + F_\times^2)(\theta, \phi)$ of an ideal interferometric gravitational wave detector with arms $\hat{e}_x = \hat{x}$ and $\hat{e}_y = \hat{y}$ 86

4.2 Antenna patterns of existing detectors (a) LIGO Hanford (both instruments), (b) LIGO Livingston, (c) VIRGO, (d) GEO, (e) TAMA and proposed detector (f) AIGO. 89

5.1 Relative merit of an additional site to augment the LIGO Livingston Observatory in a coincident analysis (lighter is better, contours every 2.5%). The minimum detection rate is 41% of the maximum. 101

5.2 Relative merit of an additional site to augment the LIGO Livingston Observatory in a coherent analysis (lighter is better, contours every 2.5%). The minimum detection rate is 89% of the maximum. 102

5.3 Relative merit of an additional site to augment a network consisting of the LIGO Hanford (4km) Observatory, the LIGO Livingston Observatory and a 4km LIGO I instrument at the VIRGO site, in a coincident analysis (lighter is better, contours every 2.5%). The minimum detection rate is 69% of the maximum. 103

- 5.4 Relative merit of an additional site to augment a network consisting of the LIGO Hanford (4km) Observatory, the LIGO Livingston Observatory and a 4km LIGO I instrument at the VIRGO site, in a coherent analysis (lighter is better, contours every 2.5%). The minimum detection rate is 94% of the maximum. 104
- 6.1 Sidereally-averaged response to a uniform distribution of pulsars of interferometers with varying latitudes and orientations—effectively the familiar peanut antenna pattern averaged over a rotation. The responses are independent of longitude; the vertical axis of the diagram is the Earth’s axis of rotation. From left to right, latitudes of 0° , $\pm 30^\circ$, $\pm 60^\circ$ and $\pm 90^\circ$. From top to bottom, orientations of 0 , $\frac{\pi}{8}$ and $\frac{\pi}{4}$ from north. Note that for the equatorial detector with a $\frac{\pi}{4}$ orientation, an antenna pattern null aligns with the Earth’s axis of rotation so that no sources from that direction could be detected. 111
- 6.2 Model for the distribution of galactic pulsars, in celestial coordinates. 113
- 6.3 Detectable fraction (vertical) of a galactic pulsar population against relative detection (horizontal) threshold for various detector latitudes and orientations (lines). The latitude and orientation have a minimal effect on an detector’s ability to observe galactic neutron stars, which is almost wholly governed by its baseline strain sensitivity. 114

List of Listings

| | | |
|------|---|----|
| 2.1 | Example LDAS job. | 12 |
| 2.2 | Universal Data Type class definition. | 16 |
| 2.3 | UDT cast definition. | 17 |
| 2.4 | An impossible Scalar definition | 19 |
| 2.5 | Scalar<T> definition. | 19 |
| 2.6 | Scalar<T> conversions. | 20 |
| 2.7 | Sequence<T> definition. | 21 |
| 2.8 | Sequence<double> functionality. | 22 |
| 2.9 | Matrix<T> definition. | 22 |
| 2.10 | Matrix arithmetic operation. | 23 |
| 2.11 | Matrix proxy class | 23 |
| 2.12 | Matrix functionality. | 24 |
| 2.13 | Matrix functionality. | 24 |
| 2.14 | State class definition | 25 |
| 2.15 | MixerState class definition | 27 |
| 2.16 | Mixer class definition | 27 |
| 2.17 | template method Mixer::apply definition | 28 |
| 2.18 | UDT method Mixer::apply | 29 |
| 2.19 | Helper template method Mixer::applyAs | 29 |
| 2.20 | LinFilt class definition | 32 |
| 2.21 | LinFiltState class definition | 32 |
| 2.22 | Resample class definition. | 34 |
| 2.23 | ResampleState class definition. | 35 |
| 2.24 | Valid data conditioning API commands. | 36 |
| 2.25 | CallChain class definition. | 37 |
| 2.26 | CallChain::Function definition. | 38 |
| 2.27 | MixFunction class definition. | 38 |
| 2.28 | Mix action evaluator. | 39 |

| | | |
|------|---|-----|
| 2.29 | Mixer class unit test. | 41 |
| 2.30 | Mix action (MixFunction class) unit test | 42 |
| 2.31 | LDAS mixing test job. | 44 |
| 4.1 | Compute the response matrix of a gravitational wave detector. | 87 |
| 4.2 | Compute the polarisation basis of a source. | 88 |
| 4.3 | Compute the response of a detector to a given strain. | 88 |
| 4.4 | Compute the response of a real observatory. | 90 |
| A.1 | BandSelector class definition. | 121 |
| A.2 | OEModel interface. | 124 |
| A.3 | OEModel interface (continued). | 125 |
| A.4 | OEModel interface (continued). | 125 |
| A.5 | OEModel interface (continued). | 126 |
| A.6 | OEModel interface (continued). | 126 |
| A.7 | OEModel interface (continued). | 127 |
| A.8 | OEModel interface (continued). | 127 |
| A.9 | OEModel state abstraction. | 127 |
| A.10 | OEModel state implementation. | 128 |
| A.11 | OEModel state implementation (continued). | 129 |
| A.12 | Progressive model estimator. | 129 |
| A.13 | OEModel linear filter implementation. | 130 |
| A.14 | LineRemover interface. | 131 |
| A.15 | LineRemover implementation. | 134 |
| B.1 | Siderially-averaged response | 151 |

Chapter 1

Introduction

If you need to use statistics, you ought to have done a better experiment.

— Lord Rutherford

If only it was that simple. Hundreds of millions of dollars have been poured into the construction of phenomenally sensitive laser interferometers, kilometres long, pushing forward the state of the art in a number of fields and breaking a litany of world records. Yet even with some of the most sensitive instruments ever built, the gravitational waves we expect to receive will still be at the very margins of detectability. Though sensitivities will improve, the vast bulk of signals will remain in this marginal regime for the foreseeable future. We cannot merely do a better experiment.

This leaves us with a need to use statistics. With the interferometers orders of magnitude more sensitive to things as diverse as the alternating current electrical supply, the moon, artillery exercises and tree-felling, than to the infinitesimal gravitational waves emitted by colliding black holes in nearby galaxies, the potential for false positives is immense—and unaffordable for a field whose very history begins with unsupportable claims. The leading US-based Laser Interferometer Gravitational wave Observatory (LIGO) has gone to the extraordinary length of building two vast twin instruments at opposite corners of the continental United States, so that each may independently validate any detection by the other. The raw output of the instruments must be carefully vetoed, conditioned, filtered and compared to efficiently detect a signal—and all with a thorough understanding of how these many processes affect the statistics of, and our confidence in, the detection. Even with such extraordinary precautions, it is possible that

there may be no single, widely agreed-upon moment of discovery—only a growing confidence, perhaps analogous to the recent process of discovery of extra-solar planets.

In this context, this thesis addresses several issues involved in improving the sensitivity of, and confidence in, the results of the emerging field of gravitational wave astronomy. The first is the author’s contribution to the LIGO Data Analysis System (LDAS), including the design, implementation, and—most importantly—validation of primitive signal processing operations that will condition much of LIGO’s data output. The second is the development and—critically—characterisation of a specific data conditioning technique to remove spectral line interference. The third consists of the development of models of the collective sensitivity of a global collaboration of gravitational wave detectors, to determine the sensitivities of existing and future configurations to a variety of possible sources, and to develop recommendations as to the configuration and expansion of the network. In the latter, the possible contribution of an Australian gravitational wave observatory is given special consideration.

1.1 Review

In the ‘weak-field limit’ of general relativity where only small perturbations $h_{\mu\nu}$ to the metric of flat space-time $\eta_{\mu\nu} = \text{diag}(-1, 1, 1, 1)$ are considered, the Einstein equations can be linearised. If we choose to work in the *transverse traceless gauge*, where the coordinate system is defined by the trajectories of freely-falling test particles, the linearisation produces a *wave equation*,

$$\left(\nabla^2 - \frac{1}{c^2} \frac{\partial^2}{\partial t^2} \right) h_{\mu\nu} = 0, \quad (1.1)$$

representing a plane wave of space-time curvature propagating at the speed of light c .

The scale of gravitational waves can be estimated by considering the field equation $\mathbf{T} = \frac{c^4}{8\pi G} \mathbf{G}$ in analogy with Hooke’s law $P = Eh$. The ‘stiffness’ of space-time, $\frac{c^4}{8\pi G}$, is a vast number; correspondingly, a large stress-energy density results in only a small curvature of space-time. Plausible astrophysical sources of gravitational waves produce a *strain* of only $h \approx 10^{-21}$ at the Earth; possible laboratory sources are much weaker, only $h \approx 10^{-39}$.

Only a literally astronomical expenditure of energy can produce a detectable gravitational wave.

Until relatively recently, debate raged about the physical reality of gravitational waves, as to whether any experiment could actually detect them. Any such experiment would be embedded in space-time, and from the definition of the transverse traceless gauge used to formulate the problem, the passage of a wave will not alter the coordinates of any test particle in the experiment. However, it is not the coordinates of test particles but rather the measured distance $\int \sqrt{g_{\mu\nu} dx^\mu dx^\nu}$ between two freely falling particles that will be changed as the space-time metric $g_{\mu\nu} = \eta_{\mu\nu} + h_{\mu\nu}$ is perturbed. Gravitational wave detection consists of making this measurement.

The first generation of gravitational wave observatories were *resonant mass detectors*, or *Webber bars*. Webber noted that a gravitational wave passing through a solid body would change its length, and if the body was resonant with the gravitational wave, each cycle would coherently add until a detectable amplitude resulted. Although Webber's claims to have observed gravitational radiation are discredited, over the intervening decades Webber bars have grown in size, sophistication and sensitivity—now approaching the 10^{-21} level in narrow frequency bands—but they have not yet made any detections.

The current generation of gravitational wave detectors is very different: based, in a pleasing piece of historical serendipity, on the Michelson interferometer, whose disproval of an aether drift was one of the cornerstone pieces of experimental evidence for relativity.

A Michelson interferometer consists of a light source (in modern instruments, a coherent laser light source), a beam-splitter and two mirrors. Light enters the beam-splitter, and is split into two beams. Each beam travels down an 'arm' of the interferometer, is reflected by an end mirror back to the beam-splitter. The beam-splitter recombines the returning coherent light, producing interference fringes. These fringes depend sensitively on the difference in the lengths of the two arms—to a fraction of a wavelength of the light—making the Michelson interferometer a sensitive measure of distance.

To detect gravitational waves with frequencies from 30 to 1000 Hz, the optimal arm length of a Michelson interferometer is more than 100 km, so that the light travel time is less than, but comparable to, (half) the period of the wave, and the relative change in length induces as large an absolute change in arm length as possible. Such an instrument cannot reasonably be constructed on the curved surface of the Earth. However, the arms can

be ‘folded’ by adding an additional mirror in each arm to reflect the light back and forth in the arm for either a fixed (delay line) or statistical (optical cavity) amount of time, reducing the current generation of interferometers to more manageable—but still challenging—arm lengths of a few kilometres.

The limiting factors on such a large and sensitive instrument range from the profound (numerous quantum effects such as the uncertainty in the position of the mirrors) to the absurd (seismic noise from road bumps), and every attempt is made to limit their effect.

However, once the signal from the output photodetector has been digitised and recorded, the problem of gravitational wave detection shifts from one of engineering and experimental physics to the nascent field of data analysis. The hard-won output of gravitational wave detectors must be scrubbed of any remaining identifiable noise sources, and scoured for the faintest of gravitational wave signals. This thesis addresses the problems of how to remove a particular class of environmentally correlated noise from the output of gravitational wave detectors, and, looking to the future, examines how best to combine the output of detectors around the globe as the field moves from gravitational wave detection to gravitational wave astronomy.

1.2 Publications

Much of the work in this thesis relates to previous publications of its author. Specifically, Chapter 3 expands upon (2, 4 and 14), and Chapters 4, 5 and 6 expand upon (3). (1, 5, 6, 8 and 9) contain information on the Australian gravitational wave effort, and in particular the local establishment of a LIGO Data Analysis System (LDAS) as described in Chapter 2. I am honoured to have participated in the analysis for, and to be one of the many authors of, the first LIGO observational results (7 and 10–14).

1. D McClelland, S Scott, M Gray, D Shaddock, B Slagmolen, **A Searle**, D Blair, J Lu, J Winterflood, F Benabid, M Baker, J Munch, P Veitch, M Hamilton, M Ostermeyer, D Mudge, D Ottaway, C Hollitt, “Second-generation laser interferometry for gravitational wave detection: ACIGA progress”, *Class. Quant Grav.* **18** 4121–4126 (2001).
2. S Scott, D McClelland, **A Searle**, P Charlton, B Whiting, “A Gaussianity measure for laser interferometer data”, *Proc. Ninth Marcel*

- Grossmann Meet. Gen. Rel.* (eds. V Gurzadyan, R Jantzen, R Ruffini), World Scientific Singapore, 1919–1920 (2002).
3. A Searle, S Scott and D McClelland, “Network sensitivity to geographical configuration”, *Class. Quant Grav.* **19** 1465–1470 (2002).
 4. A Searle, S Scott and D McClelland, “Spectral Line Removal in the LIGO Data Analysis System (LDAS)”, *Class. Quant Grav.* **20** S721–S730 (2003)
 5. J Jacob *et al*, “Australia’s Role in Gravitational Wave Detection”, *Pub. Astro. Soc. Aust.* **20** 223–241 (2003)
 6. M Gray *et al*, “ACIGA: Status Report”, *Proc. SPIE* (Gravitational-Wave Detection; eds. M Cruise, P Saulson) **4856** 258–269 (2003)
 7. B Allen *et al*, “Upper limits on the strength of periodic gravitational waves from PSR J1939+2134”, *Class. Quant Grav.* **21** S671–S676 (2004)
 8. S Scott, A Searle, B Cusack and D McClelland, “The ACIGA Data Analysis Programme”, *Class. Quant Grav.* **21** S853–S856 (2004)
 9. L Ju *et al*, “ACIGA’s high optical power test facility”, *Class. Quant Grav.* **21** S887–S893 (2004)
 10. B Abbott *et al*, “Detector Description and Performance for the First Coincidence Observations Between LIGO and GEO”, *Nuc. Inst. A* **517**, 154–179 (2004)
 11. B Abbott *et al*, “Setting upper limits on the strength of periodic gravitational waves from PSR J1939+2134 using the first science data from the GEO 600 and LIGO detectors”, *Phys. Rev. D* **69**, 082004 (2004)
 12. B Abbott *et al*, “First upper limits from LIGO on gravitational wave bursts”, *Phys. Rev. D* **69**, 102001 (2004)
 13. B Abbott *et al*, “Analysis of LIGO data for gravitational waves from binary neutron stars”, *Phys. Rev. D* **69**, 122001 (2004)
 14. B Abbott *et al*, “Analysis of first LIGO science data for stochastic gravitational waves”, *Phys. Rev. D* **69**, 122004 (2004)

1.3 Overview

Part I deals with the software infrastructure for the analysis of LIGO data throughout this decade. Chapter 2 gives an overview of the LIGO Data Analysis System (LDAS) and details the implementation and validation of the basic elements of the *data conditioning API*, including its primitive signal processing operations. Chapter 3 considers the development of a data conditioning tool to remove spectral line interference: its implementation within the data conditioning API using the functionality of Chapter 2, validation of its correctness, and characterisation of its ability to improve the recovery of gravitational wave signals.

Part II deals with the simulation of global networks of gravitational wave detectors. Chapter 4 presents a basic formalism for the analyses. Chapter 5 determines the optimal location for new observatories to supplement existing networks, for the case of a uniform population of binary inspiral events detected by coincident or coherent analysis. Chapter 6 considers the effects of observatory latitude on the detector's sensitivity to a population of galactic neutron stars.

Part I
Data Conditioning

Chapter 2

The LDAS data conditioning API

Shouldn't acronyms be expanded in titles?

— Katie Weir

The light amplification by stimulated emission of radiation interferometer gravitational-wave observatory data analysis system data conditioning application programming interface?

— Antony Searle

This chapter is intended to serve a dual purpose: first, to familiarise the reader with the elements of the data conditioning API utilised by the implementation of the line removal technique in Chapter 3; second, to provide a guide to the ‘design and evolution’ of the data conditioning API for its future maintainers and contributors. The elements referred to in Chapter 3 are largely among those to which the author made substantial contributions, and, happily, are illustrative of the system as a whole.

We begin with a brief survey of LDAS, noting in particular the position and function of the data conditioning API within it, followed by a discussion of the motivating factors behind the data conditioning API itself. The development has proceeded in parallel with the commissioning of the LIGO instruments, meeting milestones to facilitate engineering and science ‘runs’, including ‘Mock Data Challenges’ (MDCs) consisting of integration and stress testing.

The code listings presented in this chapter have been edited (often extensively) for clarity. Unless otherwise noted, the original code is publicly available from the LDAS repository [1]. Brief notes on the C++ language have been included in the text as an aid to interpreting the listings; for an authoritative introduction to the C++ language, we recommend [2].

Disclaimer The author is a significant, but not the sole or even primary contributor to the data conditioning API, which is a collaborative effort between many contributors from LSC institutions, coordinated by the California Institute of Technology.

2.1 The LIGO Data Analysis System (LDAS)

To search for gravitational wave signals in noisy data *optimally* (in the theoretical sense) is, in general, computationally expensive. In practice, approximately optimal methods are used; their sensitivity is limited by available computational cycles, memory or network bandwidth. As the instruments continuously acquire data, this processing must occur in real time, or an unmanageable backlog will result.

The need for continuous real-time computation ruled out the use of general purpose supercomputing resources. Instead LIGO elected to develop a custom system to run on its own hardware.

An LDAS implementation runs on a heterogeneous network of Sun or x86 Linux servers coupled to a Beowulf cluster of inexpensive commodity PCs. LDAS is responsible for acquiring on the order of a terabyte of information from the observatories daily and caching it locally, producing a reduced data set (RDS) for transfer to an archive and other LDAS implementations, and pre-processing the data and managing the specific parallel codes that search the data for gravitational wave signals.

In the course of this project, an LDAS implementation has been maintained at The Australian National University (ANU) by the author, first on hardware loaned from the Pennsylvania State University, and currently on the ACIGA Data Analysis Cluster (ADAC), a purpose-built local LDAS system.

2.1.1 APIs

LDAS software is implemented as a number of Application Programming Interface (API) libraries of compiled C++ code, which are called by interpreted TCL (Tool Command Language [3]) script drivers. The term ‘API’ has also come to refer to the process relying primarily on a particular API library. Each API typically resides on a different machine; they communicate with each other over a network.

LDAS jobs are typically configured as a ‘pipeline’. Data is read into the system by one API, then passed along to another that processes it and passes it along to another API.

The *manager* API oversees the LDAS system. It receives incoming user commands and passes them on to the relevant API. It is responsible for starting and stopping the other APIs and regularly polls their state, restarting any that have crashed or become unresponsive.

The *disk cache* API maintains a list of the data files available to the LDAS system.

The *frame* API responds to queries for data specified by time and channel, and queries the disk cache API for the pertinent files. It then reads the files to extract segments of the channels, concatenates the data together, and, if necessary, converts the data to the requested sampling rate, before passing it on.

The *data conditioning* API processes data received from the frame API before passing it on to the wrapper API (or another target). It uses a flexible command language to apply a broad range of built-in signal-processing actions.

The *wrapper* API manages the parallel search codes, providing a ‘wrapper’ around an off-the-shelf implementation of the Message Passing Interface (MPI) [4] standard for parallel computation. It provides assistance for the common tasks of input, output and load-balancing.

(There are other APIs—most notably the database-wrapping *metadata* API—that do not concern us at this time.)

2.1.2 Command language

Once an LDAS system has been successfully initialised by the manager API, the manager API will listen for incoming ‘jobs’, then invoke the appropriate APIs to perform the job. Each job contains a username and password for

authentication purposes (the manager API maintains its own structure of user accounts and permissions), and the email address of the user, to which a message containing (the URL to) the results of the job (or a diagnostic message) are sent. Often, the ‘user’ is in fact a driver program. (Jobs are often written as TCL scripts; when executed they can substitute in values derived from arguments or environment variables and then submit themselves.)

Listing 2.1: Example LDAS job.

```
ldasJob {
  -name acsearle
  -password *****
  -email acsearle@localhost
} {
  dataPipeline
  -framequery {
    { R H {} 714975000-714975001 Adc(H1:LSC-AS-Q!resample!8!) }
  }
  -output { ilwd ascii }
  -aliases { h1 = H1; }
  -datacondtarget { datacond }
  -algorithms {
    x = slice(h1, 0, 1024, 1); # data, start, count, stride
    output(x,-,x,x,x);
  }
}
```

The type of job (in this case, `dataPipeline`) dictates which named options are required.

The `-framequery` option lists the data that will be required by the job. (The data type is `R`, ‘raw’, and the observatory is `H`, Hanford.) No files, `{}`, are specified; these will be automatically determined by the frame API. The ‘era’ requested is the 1 second interval between GPS times 714975000–714975001. The frame API will seek the `Adc` (analog-to-digital) channel named `H1:LSC-AS-Q` (the output of the 4 km interferometer); the postfix `!resample!8!` instructs the frame API to downsample the channel by a factor of 8 (in the case of `H1:LSC-AS-Q`, from 16384 Hz to 2048 Hz).

The `-aliases` option assigns human-readable aliases to the often-long channel names. In this case the concise alias ‘`h1`’ is associated with the channel whose name contains the substring ‘`H1`’.

The target for the output of the data conditioning API is the data conditioning API itself, `-datacondtarget { datacond }`, which causes its output to be written to disk. (To conduct searches the output would be sent to the wrapper API.)

The `-output` format will be `ilwd ascii`, a human-readable (ASCII-encoded) version of the XML-derivative Internal LightWeight Data (ILWD) format used by LIGO.

The `-algorithms` option is where the data conditioning API command language is embedded. It consists of a series of C- or MATLAB-like statements forming a fully programmable signal-processing pipeline. (In this case, it slices off the first 1024 elements of the input data into the variable `x` and outputs it.) It is the implementation of this functionality that will concern us for the rest of this chapter.

2.2 Design and evolution

The original design for the data conditioning API is encapsulated in [5]. The seeds are to be found there of the issues that would come to dominate its development.

The data conditioning API library is written in ISO C++. The rationale given in [5] mentions its C heritage, the (then recent) ANSI/ISO standardisation of the language, and the benefits of its object-oriented features. The primary motivation is one of efficiency: C++ is a compiled language, with mature optimisers, and typically runs faster and uses less memory than interpreted languages. Moreover, it permits higher-level abstractions (and thus has better prospects for reuse and maintenance) than C while maintaining, by design, '95%' of C's performance; this is a consequence of (its designer) Stroustrup's "you don't pay for what you don't use" philosophy [2] and was perhaps the leading factor in its widespread adoption. (Counter-intuitively, the combination of C++'s **template** and **inline** features can produce code that is faster than C, by shifting some computation from run-time to compile-time [6].)

Though the standardisation of C++ represented a great step forward, the standard itself differed significantly from existing practice.¹ Some useful and significant parts of the standard proved pathologically difficult for

¹ANSI/ISO C++ committee member Bruce Eckel later referred to the process as language design "by thought experiment".

vendors to implement. It was not until late 2002 that a compiler was finally released that could reasonably claim compliance to the standard; the difficulties experienced by its implementers have been held up as reason to amend certain features in the next iteration of the standard (the C++-0x standard, expected sometime in the middle of this decade). Throughout the period of development discussed here (2000–2003 inclusive) the GNU C++ compiler used by LDAS has improved dramatically, but much of the code base has been impaired by this shifting foundation. (Perhaps the most obvious sign of this is the persistence of the once-necessary evil of explicit **template** instantiations.²)

Compounding the problem of a changing language implementation, the C++-97 standard also spawned a change in the C++ programming idiom. Meyers laments in his foreword to Alexandrescu [7] that the C++ community’s understanding of **templates** has been undergoing “dramatic change” for a decade. Similar changes in thinking on exception specifications and other features have also occurred. These new features and idioms could have been helpful in the design and implementation of LDAS, but most were not even conceived of, let alone widely known, at that time. (Many of these techniques appear in [7] and the useful Boost library [8], some parts of which are proposed for inclusion in C++-0x.)

The LDAS model of an API as a compiled library driven by an interpreted script has several benefits. The library performs specific well-defined tasks quickly and efficiently at the cost of flexibility (in the sense of rapid development). Complementarily, the script may be rapidly altered to make use of that functionality in different ways.

The data conditioning API inherited this two-tiered model from the design of LDAS as a whole, but in its particular case the model caused problems. This framework is most applicable when the functionality of the library is stable, and the interface it presents to the script is correspondingly stable, concise, and well-defined. The requirements placed on the data conditioning API meant that these conditions were not consistent with reality. The generality of the operations required by the data conditioning API—effectively the implementation of a numerically-oriented programming language somewhat like MATLAB—caused an explosion in the number of data types³ and

²*Not* a search template!

³A data type can be thought of as a collection of bits, *and* assertions about what the bits represent. For example, 32 identical bits may represent different numbers depending on if their type is **int** (integer) or **float** (floating point real). Much of C++’s power is

operations that had to be implemented in C++ and exposed to and handled by the scripting layer.

It was realised at the first Mock Data Challenge that this model of development was unsustainable. The first step in fixing the problem was the rapid development of a ‘universal data type’, so that only one type needed to be exposed to the scripting layer. It was followed by the adaptation of existing signal-processing operations to a new call-chain framework that is effectively a simple *virtual machine*. The involvement of the scripting layer in the implementation of the data conditioning pipeline was reduced until it merely parsed input (it still performs an important role in initialisation and output).

The problem of data typing, or more specifically, the problem of how to efficiently implement the un-typed data conditioning API command language in strongly typed C++, came to dominate the implementation, and has not yet been adequately addressed. Development at the moment relies at least partially on the brute-force handling of the multiplicity of combinations of operations and data types. This leaves the implementation with a fundamental scalability problem.

It is not readily apparent (to the author, at least) at this time how the original design could have been implemented in a significantly superior way. Radical changes in the design could be supposed to fix some or all of the problems that were encountered, but there would be no guarantee that they would not introduce their own issues. Part of the problem is due to the well-known issue of *multiple dispatch* (run-time polymorphic behaviour on more than one type), an issue not addressed by C++ (or most modern languages). The adoption of more modern C++ techniques would undoubtedly improve conciseness and reduce repetition, but would not in itself address the fundamental scalability issue. No such proposal has yet merited the large-scale overhaul of what is, after all, a working code base.

It should be noted that the above discussion is in some sense an argument over aesthetics (though it does affect the practicality of future expansion). The problems are not apparent to the user, only the developer. In particular, a stringent testing regimen ensures that the results produced by the data conditioning API are correct.

obtained from utilising the assertions provided by type information at compile time to avoid the expense of considering those assertions at run time.

2.3 Universal Data Type (UDT)

The Frame data format of LIGO and Virgo, and the ILWD format of LDAS, support a wide range of data types: 8-, 16-, 32- and 64-bit integers, 32- and 64-bit floating point real numbers, and 64- and 128-bit complex numbers (composed of pairs of reals). The data conditioning API must be capable of operating on all of them, either singly (scalars) or in (homogeneous) arrays (vectors, matrices, and higher dimensional arrays). This means that LDAS must support over a dozen types. When metadata is considered—for example, decorating a vector with information declaring it to be a time series of a particular sample rate and start time—the number of types becomes immense.

The ‘universal data type’, or UDT, is intended to hide all these types behind a common interface. Only those operations which need to know the exact type of a UDT will have to look beyond the interface. Much of the data conditioning API implementation can remain oblivious to the exact nature of the data flowing through it.

2.3.1 Implementation

The inheritance mechanism of C++ is exactly the mechanism required for the implementation of UDT. The UDT is a *base class* exposing those operations common to its *derived classes* (the data types). Each data type ‘is-a’ UDT, as a sedan ‘is-a’ car; they are said to *inherit* from it.

The interface of UDT is the intersection of the interfaces of all its possible derived classes, and is necessarily ‘thin’, consisting only of creation, destruction, copying, and methods to assist in the resolution of the derived class.

Listing 2.2: Universal Data Type class definition.

```
class UDT
{
public:
    virtual ~UDT(); // destroy
    virtual UDT* Clone() const = 0; // (deep) copy
    template<class T> static bool IsA(const UDT& In); // is it really
        type T?
```

```

template<class T> static T& Cast(UDT& In); // access it as a type
    T (or throw an exception)
    // ... (omitted)
};

```

In this listing, the first line opens the definition of **class** UDT, circumscribed by braces `{...}`. The keyword **public** declares all which follows to be universally accessible (the default is **private**, meaning that only the class implementation itself has access).

The class itself (in the simplified version we present) consists only of methods, and has no data members. A *method* is a function that is a member of the class; its full name is prefixed with the class name, as in `UDT::Clone`, and it can only be invoked on an *instance* of UDT, as in `my_udt.Clone()`, in which the method receives `my_udt` as the implicit ‘zeroth’ argument **this**. (The **static** keyword indicates a method that is associated with a class, but is *not* invoked on an instance of that class.)

The *virtual destructor* **virtual** `~UDT()` cleanly destroys the UDT instance on which it is invoked. The **virtual** keyword indicates that the method may (should) be *overridden* by derived classes; derived classes will replace the UDT method with a method that can cleanly destroy the particular implementation of that derived type. Even when the UDT is destroyed in a context that knows nothing about the derived type, the correct method will be called.

The *deep copy method* **virtual** `UDT* Clone()const = 0` makes a copy of the UDT that is aware of the particular derived type, just as is the virtual destructor is. The method returns a pointer, `UDT*`, to a copy of the derived type instance created by the **new** operator. The **const** keyword indicates that the original is unaffected by the copying process. The `= 0` syntax indicates that the method may not be supplied by UDT; this makes the method *pure virtual* and the UDT class *abstract*—UDT cannot be instantiated, and derived types *must* override `Clone`. (Ideally the virtual destructor should also be pure.)

The **template**<class T> methods `IsA` and `Cast` are examples of *generic* code. Many data types may inherit from UDT; we cannot (and should not) write specific code to check each case. Instead we note that the code to check a particular case differs only in the particular type for which we wish to check. The **template** keyword allows us to write a generic definition that works for *all* types by writing it in terms of the *template parameter* ‘T’. (The **dynamic_cast** keyword performs the validation and cast.)

Listing 2.3: UDT cast definition.

```

template<class T> T& UDT::Cast(UDT& a) \
{
    return dynamic_cast<T&>(a); \
}

```

The `IsA<T>` method allows us to check if a UDT instance is really an instance of class `T` (returning a **boolean true** or **false**), and the `Cast<T>` method allows us to access the class `T` instance itself. If we attempt to perform a *bad cast*—if the UDT is not a `T`—a **bad_cast** exception is **thrown** at runtime. This can be avoided by checking first with `IsA<T>`.

Critique

The UDT as presented above is textbook object oriented programming (OOP). As such, it is open to the criticisms of C++’s implementation of OOP. Clone’s use of *raw* pointers is potentially dangerous: the innocuous statement ‘`my_udo.Clone();`’ is valid but results in a memory leak. The default and copy constructors should be **protected** (so that only derived classes may call them) and the assignment operator explicitly left undefined as their default behaviour will make *slicing* (copying or assigning a more derived class to a less derived class) possible [9]. Other implementations of OOP, for example Java, already have a universal base class from which *all* classes are defined. C++ eschews this approach as introducing unacceptable overhead; the slicing and pointer issues arise from similar considerations.

The formulation of `IsA` and `Cast` as **static** methods taking a UDT argument rather than plain methods or helper functions is bizarre; however, at one time it was a necessary work-around for an equally bizarre compiler error.

A ‘better’ UDT would probably adopt the *pointer-to-implementation* (‘*pimple*’) idiom. The current arrangement would be wrapped by a new handle class with conventional copying semantics, eliminating the problem of the user dealing directly with slicing and pointers. The handle could be made *smart* [7, 10] to permit *lazy* copying, eliminating an issue we will encounter in §2.4.

2.3.2 Scalar

To represent a scalar value, one must derive a class from UDT. There are many kinds of scalar values—integers, reals, and complex numbers, repre-

sented at various precisions—yet they all have the same basic requirements. We can represent this commonality using generic coding, templatising `Scalar` on an unknown type `T`.

The design of `Scalar` presents an interesting problem. One choice denied to us is to inherit from the unknown type `T`, and thus inherit its interface, so that an object of type `Scalar<T>` could be used wheresoe'er a `T` is expected.

Listing 2.4: An impossible `Scalar` definition

```
template<typename T>
class Scalar : // inherits from
    public UDT, // and
    public T
{
    // ... (omitted)
};
```

We cannot do this because at least some of the types we will use will be *basic*, such as `int` and `float`, which are not classes and cannot be inherited from. (Making basic types full classes would impose performance overhead in common cases. Java, for example, faces the same trade-off, but makes the opposite decision.) Moreover, even for the types that are classes (`complex<float>` and `complex<double>`) derivation is problematic as they were not intended to be used as base classes, do not have virtual destructors, and thus have the potential for undefined behaviour.

Instead we must rely on *implicit* and *explicit conversions* between `Scalar<T>` and the type `T`:

Listing 2.5: `Scalar<T>` definition.

```
template<typename T>
class Scalar :
    public UDT
{
public:
    explicit Scalar(T); // explicit conversion from T
    virtual ~Scalar(); // override
    virtual Scalar* Clone() const; // override
    operator T&(); // implicit conversion to T
    operator const T&() const; // implicit conversion to const T
    const T GetValue() const; // accessor
```



```

    void SetValue(const T&); // mutator
    // ... (omitted)
private:
    T m_value; // encapsulated value
};

```

The constructor accepting a `T` defines a conversion from an instance of `T` to an instance of `Scalar<T>`. As `T` matches almost *any* type (precisely, any copy-constructible type) it is declared **explicit** to prevent its implicit application by the compiler in situations where the programmer did not explicitly request it. An implicit conversion such as this would allow anything to be converted into its `Scalar` equivalent, which ‘is-a’ UDT; this would subvert the process of compile-time error checking.

The implicit conversion to a reference to the encapsulated `T` instance is defined by **operator** `T&` (and its **const** variant; which version is called depends on the context). As it is implicit, the compiler has license to convert a `Scalar<T>` to a `T` wherever it finds it expedient to do so; most commonly where a `Scalar<T>` is passed as an argument to a function expecting a `T`.

Listing 2.6: `Scalar<T>` conversions.

```

Scalar<complex<double>> z(complex<double>(-1., 0.)); // explicit
    conversion from complex to scalar
sqrt(z); // implicit conversion from scalar to complex to use complex
    square root
z.real(); // error: 'real' undeclared
complex<double>(z).real(); // conversion made explicit

```

Implicit conversion does not occur everywhere that we might hope; for example, `complex<T>` defines methods `real` and `imag` that we cannot invoke on a `Scalar<complex<T>>` without explicitly invoking the conversion.

For this reason, we also provide traditional *accessor* and *mutator* methods to get and set the wrapped value. Note that `GetValue` does not change the value, and is thus **const**; `SetValue` does change the value, and is non-**const**.

2.3.3 Sequence

The inapplicability of implicit conversion to many cases makes the encapsulation model of `Scalar<T>` inapplicable to types that define frequently used methods (`complex<T>` is itself a marginal case). Sequence types fall into this category.

The basic C++ sequence type is the array. Inherited from C, it is an inconvenient and perilous language construct. The C++ Standard Template Library (STL) supplies a wide range of container types to replace it; its `valarray<T>` [2] is targeted at numerical computation. The C++ standard allows implementers unusual latitude in the `valarray<T>` specification to facilitate the implementation of aggressive optimisation. It provides fast vector arithmetic and BLAS (Basic Linear Algebra Subprograms)-like [11] subsequences.

As `valarray<T>` is a class, we may inherit from it as well as UDT when we design `Sequence<T>`, an option which we had to reject for `Scalar<T>`. (Note that `valarray<T>` does not include a virtual destructor, so unfortunately there is the potential for undefined behaviour in the unlikely event of a user destroying a `Sequence<T>` as a `valarray<T>`.) This way, `Sequence<T>` automatically inherits most of `valarray<T>`'s interface, such as the subscript "`operator[]`" and the size method. (Unlike many OOP languages C++ supports inheritance from multiple base classes.)

Listing 2.7: `Sequence<T>` definition.

```
template <typename T>
class Sequence :
    public UDT,
    public valarray<T>
{
public:
    Sequence(size_t n); // construct with n (default) elements
    Sequence(const T& x, size_t n); // construct with n copies of x
    Sequence(const valarray<T>&); // implicit conversion from valarray
    virtual ~Sequence(); // override
    virtual Sequence* Clone() const; // override
    // ... (omitted)
};
```

Constructors are not inherited, so much of the class definition is concerned with replicating the functionality of `valarray<T>`'s suite of constructors. It is not necessary to declare the implicit conversion from `valarray<T>` **explicit** as the pattern `valarray<T>` only matches `valarrays`. The implicit conversion the other way is provided by inheritance; every `Sequence<T>` instance 'is-a' `valarray<T>` by definition.

The inherited functionality allows us to invoke all the methods of `valarray<T>` on `Sequence<T>`:

Listing 2.8: `Sequence<double>` functionality.

```
Sequence<double> a(3); // a sequence of 3 elements
a[a.size() - 1] = 1.0; // set the last element
```

2.3.4 Matrix

There is no type in the STL representing an $n \times m$ matrix; we follow the suggestion in Stroustrup [2] and implement a generic `Matrix<T>` class using a packed `valarray<T>` of nm elements.

Listing 2.9: `Matrix<T>` definition.

```
template<class T>
class Matrix :
public UDT
{
public:
Matrix(size_t rows, size_t columns); // rows-by-columns matrix
virtual ~Matrix(); // override
virtual Matrix* Clone() const; // override
size_t rows() const; // accessor
size_t columns() const; // accessor
// ... (omitted)
private:
valarray<T> m_data; // representation
size_t m_rows; // dimensions
size_t m_columns;
};
```

Internally the representation is straightforward: two `size_ts` (non-negative integers) represent the dimensions n and m of the matrix; the elements are packed into a `valarray<T>` (the packing is FORTRAN- rather than C-ordered to facilitate the use of CLAPACK [12]). The dimensions are set on construction and may be checked with the `rows()` and `columns()` accessors.

As `Matrix<T>` inherits only from `UDT` we must explicitly define much of its interface by hand. Many operations can be implemented in terms of the corresponding `valarray<T>` operations; others can be handled by CLAPACK.

Listing 2.10: Matrix arithmetic operation.

```

Matrix& operator+=(const Matrix& right)
{
    // ... (check dimensions)
    m_data += right.m_data; // use valarray +=
    return *this; // self-reference
}
// ... (arithmetic operators)

```

To support C++-style subscripting of a `Matrix<T>` instance, as in `A[i][j]`, we must implement an `operator[]` method. This is not a trivial undertaking, as `operator[]` must return a *proxy object* representing a row (or column) of the `Matrix<T>`, with its own `operator[]` that finally returns a reference to a `Matrix<T>` element. The STL class `slice_array<T>`, a proxy object for sub-array slices of `valarray<T>`, is almost ideal for our purposes, but unfortunately its copy-constructor is private, preventing any methods but those of its `friend valarray<T>` from returning it. Instead we implement member class `Matrix<T>::proxy_array`, with essentially the same interface as `slice_array<T>`, leveraging `slice_array<T>` internally, to access rows and columns of the matrix. The slice class (another `valarray<T>` helper class) stores the start index, stride, and number of elements of the array slice.

Listing 2.11: Matrix proxy class

```

class proxy_array
{
    friend class Matrix<T>;
public:
    operator const valarray<T>() const; // conversion
    proxy_array& operator=(const valarray<T>&); // assignment
    T& operator[](size_t); // subscripting
    const T operator[](size_t i) const; // const subscripting
    proxy_array& operator+=(const valarray<T>& right); // arithmetic
    // ... (arithmetic operators)
private:
    valarray<T>& m_data; // reference to matrix elements
    slice m_slice; // subarray parameters
};
proxy_array row(size_t); // reference a row
const proxy_array row(size_t) const;

```

```

proxy_array column(size_t); // reference a column
const proxy_array column(size_t) const;
proxy_array operator[](size_t); // synonym for row
const proxy_array operator[](size_t) const

```

The `proxy_array` forwards assignment and arithmetic operations to the individual elements of the matrix. The mutators `row`, `column` and `operator[]` all return `proxy_arrays` referencing the corresponding subset of the matrix; the `proxy_array` itself behaves like a `valarray<T>` whose elements are embedded in the `Matrix<T>`. Users will typically never see the `proxy_array`.

Listing 2.12: Matrix functionality.

```

Matrix<double> A(3, 3); // 3x3 matrix
A[1] = 1.; // centre row set to [1 1 1]
A[1][1] = 0.; // centre element set to 0

```

A number of non-member **operators** are also supplied for `Matrix<T>`.

Listing 2.13: Matrix functionality.

```

template<typename T> Matrix<T> operator+(const Matrix<T>&
left, const Matrix<T>& right)
{
    return Matrix<T>(left) += right; // add to a copy and return that
}
// (...) arithmetic operators

```

Higher-dimensional arrays have not yet been required (at the interface level) in the data conditioning API.

2.3.5 Metadata

Many of the series used in the data conditioning API are time series; metadata about these series, like their sampling rate and start time (from which the sample times can be computed) are useful in many contexts. In the case of a Fourier transform, the time series metadata can be used to calibrate the frequency resolution of the transformed series, which can then be stored as frequency series metadata.

The design adopted for metadata was to mix in base classes containing metadata members to existing general UDTs. For example, `Sequence<double>` and `TimeSeriesMetaData` are combined to produce the `TimeSeries<double>` template class.

Unfortunately the proliferation of ‘customised’ UDTs creates problems for those methods that have to support the metadata. In the case of operations taking more than one argument, the addition of new relevant metadata can cause an explosive increase in the number of special cases that have to be handled. Metadata has proved so time-consuming to support that relatively little of the data conditioning API uses it to its full potential.

2.4 Signal processing

Operations on UDTs are performed by a suite of classes. In the language of *Design Patterns* [13], they may adopt a *Memento* pattern to transfer an internal state between successive calls, or may encode their own state (the *Command* pattern). If the state is to be exposed to the user, that class must inherit from UDT.

Each class has a number of tasks to perform. Typically the class will provide two forms of most methods, one accepting UDTs, and another (often templatised) accepting basic (or STL) types. The former ensure that their arguments can be cast to a valid call of the latter. When many template arguments are supported (for example, a class might support **float**, **double**, `std::complex<float>` and `std::complex<double>`) this can be fairly complicated. The non-UDT methods validate their arguments against a series of preconditions. The final task is to perform a signal processing operation.

The data conditioning API works primarily in the time domain, to facilitate the continuation of operations on successive chunks of data. Some operations store their state in an external Memento class; other operations store their own state (also a UDT) as a Command object.

It was thought to be helpful to give all these operation states a common base class, to help distinguish them from more traditional UDTs. The class `State` inherits from UDT, without adding any functionality.

Listing 2.14: State class definition

```
class State :
    public UDT
{
public:
    virtual State* Clone() const = 0;
}; // class State
```

The issue of how to efficiently return UDTs from operations is a vexed one. The **return** mechanism in C++ makes a copy of the returned object. This is potentially expensive if the object is a large sequence. Furthermore, the returned object cannot be a UDT, since the copying would *slice* away the non-UDT components. If the returned object is an exactly known UDT type, slicing will not occur, but the advantages of the UDT have been lost. If a UDT pointer (UDT*) is returned, it can be ignored and leaked, which is unacceptable.

The solution of many libraries, the STL among them, is to pass the output structure, or some proxy(-ies) for it, as an argument to the operation. This solution has one drawback: it presupposes that the output type is known, and thus requires the output type deduction logic be performed by *whatever* calls the operation. This is unacceptable.

The solution adopted by the Data Conditioning API is to give operations a method of the form **void** apply(UDT*& output, **const** UDT& input). A reference to a pointer to a UDT is passed as the output. If the pointer is 0 (null), the operation is responsible for creating a new output UDT, pointed at and owned by the referenced pointer. If the pointer is non-null, the caller has asserted that they know the output type and have supplied (and probably reused) it; the operation attempts to write the output to that UDT, and throws an exception if the type is incompatible. One common usage model this supports is repeated applications of similar operations; the output can be created on first call then over-written by successive applications.

(The smart UDT handle proposed in 2.3.1 would eliminate this return problem by encapsulating the return type.)

2.4.1 Mixer

To “mix” (heterodyne) a time series with an oscillator is one of the most basic signal processing operations.

For a series x_k , the mixed series is $y_k = x_k e^{-i(2\pi f k + \phi)}$. This results in an offset of $-f$ in Fourier space. The output sequence is always complex, whereas the input may be real or complex.

The state of a mixing operation can be represented by storing the pair of real values f and ϕ . The class `MixerState` may be constructed from frequency and phase represented either as basic **doubles** or as UDTs, or copied from a single (presumably `MixerState`) UDT. Accessors are supplied for the frequency and phase. Mutators, which validate the class invariants that

$-1 \leq f \leq 1$ and $0 \leq \phi < 2\pi$, are private and used only by the creation operators and the **friend** class `Mixer`, so that the only general way to change the value of a `MixerState` is to assign another `MixerState` to it.

Listing 2.15: MixerState class definition

```
class MixerState :
public State
{
    friend class Mixer;
public:
    MixerState(const double& phase, const double& freq);
    MixerState(const UDT& phase, const UDT& freq);
    MixerState(const UDT& state);
    virtual ~MixerState()
    virtual MixerState* Clone() const;
    double GetPhase();
    double GetFrequency();
private:
    MixerState();
    void SetPhase(const double& phase);
    void SetPhase(const UDT& phase);
    void SetFrequency(const double& freq);
    void SetFrequency(const UDT& freq);
    double m_phase;
    double m_frequency;
}; // class MixerState
```

The `Mixer` class stores a `MixerState` internally; it may be (explicitly) constructed from one, and the state may be freely accessed and altered. Note the twin apply methods; one templatised on explicit `valarray<T>`s, the other accepting UDTs and resolving their types before calling the former.

Listing 2.16: Mixer class definition

```
namespace datacondAPI
{
    class Mixer
    {
    public:
        explicit Mixer(const MixerState& state);
    };
}
```



```

void apply(UDT*& out, const UDT& in);
template<typename out_type, typename in_type> void apply(std::
    valarray<std::complex<out_type> >& out, const std::valarray<
    in_type>& in)
MixerState getState() const throw();
void getState(MixerState& state) const throw();
void getState(State*& state) const
void setState(const MixerState& state) throw();
private:
    MixerState m_state;
}; // class Mixer
} // namespace datacondAPI

```

The implementation of these apply methods is instructive. The templated `std::valarray` method contains the actual implementation of the operation, computing the new values and maintaining the internal state.

Listing 2.17: **template** method `Mixer::apply` definition

```

template<typename out_type, typename in_type> void Mixer::apply(
    valarray<std::complex<out_type> >& out, const valarray<in_type
    >& in)
{
    if (!in.size())
        throw invalid_argument("Mixer::apply: Input Sequence is empty");
        // check for non-empty input
    if (out.size() != in.size())
        out.resize(in.size()); // resize the output
    double t = m_state.GetPhase();
    const double dt = m_state.GetFrequency() * LDAS_TWOPi; //
        compute the phase difference between elements
    for (int k = 0; k < in.size(); k++)
    { // (actual implementation unrolls this loop ...)
        out[k] = in[k] * complex<double>(cos(t), sin(t));
        t += dt;
    }
    m_state.SetPhase(fmod(phi,LDAS_TWOPi)); // update the state
}

```

In contrast, the UDT method is concerned with resolving types. The code

involved—little more than a “switch on type”—is exactly the kind of code railed against by C++ texts. The helper template method `applyAs` helps minimise odious duplication.

Listing 2.18: UDT method `Mixer::apply`

```

void Mixer::apply(UDT*& out, const UDT& in)
{
    bool ok = applyAs<Sequence<complex<float> >, Sequence<float>
        >(out, in);
    if (!ok)
        ok = applyAs<Sequence<complex<double> >, Sequence<double
            > >(out, in);
    if (!ok)
        ok = applyAs<Sequence<complex<float> >, Sequence<std::
            complex<float> > >(out, in);
    if (!ok)
        ok = applyAs<Sequence<complex<double> >, Sequence<complex
            <double> > >(out, in);
    if (!ok)
        throw General::unimplemented_error(" Mixer::apply: apply on
            unimplemented type");
}

```

The `Mixer::apply` method posits all the supported types and attempts to perform the mixing operation on them. `Mixer::applyAs` is responsible for the per-type testing and casting; it firsts establishes if the input type is appropriate, then ensures that the output type exists and is appropriate, and finally casts input and output to Sequences so that the concrete `apply` method may be used on them. In the case of an exception occurring anywhere, `applyAs` is careful to **catch** it and **delete** any output UDT it created.

Listing 2.19: Helper template method `Mixer::applyAs`

```

template <class Tout, class Tin> bool Mixer::applyAs(UDT*& out,
    const UDT& in)
{
    bool ok = false;
    bool null_out = ( out == 0 );
    try
    {

```

```

    if (UDT::IsA<Tin>(in)) // ensure input is the suggested type
    {
        if (null_out) // if output was not supplied
            out = new Tout; // create output
        else if (!UDT::IsA<Tout>(*out)) // if supplied output is not the
            suggested type
            throw std::invalid_argument(" datacondAPI::Mixer::applyAs<>:
                type out not compatible with type in");
        apply(UDT::Cast<Tout>(*out), UDT::Cast<Tin>(in));
        ok = true;
    }
}
catch(...) // an exception occurred
{
    if(null_out)
    {
        delete out; // destroy up the output we created
        out = 0;
    }
    throw; // rethrow the exception
}
return ok; // return success or failure (wrong input)
}

```

Critique

The implementation of Mixer is ugly. Explicit casting is unsightly; mixing exceptions and native pointers is error-prone. Numerous solutions have been considered—virtual functions, the *Visitor* pattern—but the issue is fundamentally the *multiple-dispatch* problem. C++ (and the bulk of modern languages) have no facility to make a function **virtual** on more than one type, and workarounds [7] for this issue leave much to be desired. Fundamentally, the design means the code has to be ugly somewhere.

The alternative, a major rethinking of the system to design around this limitation, is superficially attractive, but it is not clear that this would produce a solution, or that it would be reasonable to rewrite a body of tested, working code on primarily aesthetic grounds.

The use of *smart pointers* would obviously assist with issues of ownership and error management; however, at the time of development the idiom was relatively new and issues of thread safety were raised.

Another more fundamental problem with the design is the arbitrary—and in hindsight unnatural—decision to separate the mixer and its state, producing a class with methods but no non-trivial data, and a class with data but no non-trivial methods. This is a subversion of the spirit of object-oriented programming, and results in nothing but needless code proliferation. The logical next revision of the design would be to move the `Mixer::apply` methods to `MixerState`, remove `Mixer` altogether, and rename `MixerState` to `Mixer`. The amount of working, if unsightly, code that would be broken by this primarily aesthetic revision make it unfeasible.

2.4.2 LinFilt

Another fundamental signal-processing operation is linear filtering. More typically performed in the frequency domain (and for good reason), time-domain linear filtering gives the Data Conditioning API the ability to seamlessly filter consecutive time-domain sequences, but introduces some interesting effects.

A time-domain causal linear filter is defined by a series of coefficients, $a_1 \dots a_n$ and $b_1 \dots b_m$. Applying such a filter to a discrete series $u(t)$ produces a series $y(t)$ where

$$y(t) = -a_1y(t-1) - \dots - a_ny(t-n) + b_1u(t-1) + \dots + b_mu(t-m). \quad (2.1)$$

Filters for which the $a_i = 0$ depend only on the values of $u(t)$ for the samples $t-m, \dots, t$, and are known as Finite Impulse Response (FIR) filters, as the response to the input for any one time will affect only a finite portion of the sequence $y(t)$. Nonzero a_i allows the sequence $y(t)$ to depend on its own past state, permitting an Infinite Impulse Response (IIR). Despite their name, the IIR filters we will be concerned with have the property that their dependance on a sample decreases rapidly (typically exponentially) as the time since that sample increases. (Such IIR filters can be thus be arbitrarily well approximated by FIR filters.)

When the input series $u(t)$ is finite, as is the case for all data handled in the data conditioning API, $u(t)$ for $t < 0$ is unknown, and by convention set to zero. Yet the dependance of, for example, $y(0)$ on $u(-m) = 0$, means that

this convention affects the samples produced by the filter. The result is a *start-up transient* of the same duration as the impulse response of the filter.

Listing 2.20: LinFilt class definition

```

class LinFilt
{
public:
    LinFilt(const LinFiltState& state);
    LinFilt(const valarray<double>& b, const valarray<double>& a);
    LinFiltState getState();
    void getState(LinFiltState& state);
    void getState(UDT*& state)
    void setState(const LinFiltState& state);
    template<class T> void apply(valarray<T>& x)
    template<class TOut, class TIn> void apply(valarray<TOut>& out,
        const valarray<TIn>& in)
    void apply(UDT*& out, const UDT& in)
private:
    template <class T> bool applyAs(UDT*& out, const UDT& in)
    bool asSequence(UDT*& out, const UDT& in)
    bool asTimeSeries(UDT*& out, const UDT& in)
    LinFilt();
    LinFiltState m_state;
};

```

One LinFilt constructor accepts two valarray<double>s representing $a_1 \dots a_n$ and $b_1 \dots b_m$. The others are concerned with the LinFiltState memento, in UDT or resolved form. Like Mixer, the apply method is overloaded to accept (untyped) UDTs or (typed) valarrays.

Listing 2.21: LinFiltState class definition

```

class LinFiltState :
    public State
{
public:
    LinFiltState(const Sequence<double>& b, const Sequence<double>
        & a = Sequence<double>(1.0, 1));
    LinFiltState(const Sequence<complex<double> >& zeroes, const
        Sequence<complex<double> >& poles, const double& gain);

```

```

Sequence<double> LinFiltState(const UDT& b, const UDT& a);
LinFiltState(const UDT& b_or_state);
void getB(Sequence<double>& b) const;
void getA(Sequence<double>& a) const;
void getSize(int& aSize, int& bSize);
virtual LinFiltState* Clone() const;
template<class T> void apply(valarray<T>& x);
template<class TOut, class TIn> void apply(valarray<TOut>& y,
    const valarray<TIn>& y);
private:
void checkAB() const;
Sequence<double> m_b;
Sequence<double> m_a;
auto_ptr<Filters::LinFiltBase> m_linfilt;
}; // class LinFiltState

```

The linear filter is not a trivial implementation of Equation 2.1. Instead, the filter maintains an internal ‘stack’ z_i whose length is $\max(m, n)$, initialised to zero. When an element of u is read in, $b_i u_i$ is added to the value of z_i . The corresponding element of y is given by z_1 . Then the stack is ‘shifted’ so that $z_i \rightarrow z_{i-1}$, and then ya_i is added to the value of z_i . This implementation has the virtue that it does not need the random-access to previous elements of the sequence implied by the form of equation 2.1.

(The core filtering code proved so useful that it was moved to the general library, so it could be used on data exiting the FrameAPI, and elsewhere in LDAS.)

Most of the criticisms of the design of Mixer are equally applicable to LinFilt. Better integration with the Filters class it spawned in the general library could abrogate the need for its existence.

2.4.3 Resample

The Resample class is responsible for changing the sampling rate of a time series provided to it—for example, downsampling a 16384 Hz channel to 2048 Hz. It is not sufficient to construct a new series $y_i = x_{8i}$; power from the 1024–8192 Hz band will be *aliased* into the 0–1024 Hz band. Instead the series must first be low-pass filtered to remove power from the high band, and this low-pass filtering introduces the usual host of subtleties to the process.

For upsampling, an interpolation filter is used.

Resampling can be performed not just to integer multiples or quotients of the sampling rate, but to general rationals, provided by the upsampling and downsampling ratios, p and q .

The exact results of the resample operation depend on the filter parameters; this is equivalent to shaping the falloff of the low-pass filter.

The implementation of `Resample` combines the filtering and sampling stages to eliminate the production of an intermediate sequence and attendant waste of computations and memory.

Listing 2.22: Resample class definition.

```

class Resample
{
public:
    Resample(int p, int q, int n = 10, double beta = 5);
    Resample(const UDT& p, const UDT& q, const UDT& n = Scalar<
        int>(10), const UDT& beta = Scalar<double>(5.0));
    Resample(const UDT& state);
    Resample(int p, int q, const Sequence<double>& b);
    Resample(const Resample& rsmpl);
    ~Resample();
    void operator() (UDT*& out, const UDT& in);
    template<class T> void operator() (valarray<T>& out, const
        valarray<T>& in);
    void apply(UDT*& out, const UDT& in);
    template<class T> void apply(valarray<T>& out, const valarray<T>
        & in);
    ResampleState getState();
    void getState(ResampleState& state) const;
    void getState(UDT*& state);
    template<class T> void getState(Sequence<T>& state);
    void setState(const ResampleState& state);
    void getPQ(int& p, int& q);
    void getNBeta(int& n, double& beta);
    void getDelay(double& delay);
    double getDelay(void);
private:
    ResampleState m_state;

```

|};

|

Conceptually, downsampling is the application of a low-pass filter followed by the decimation of the result (similarly upsampling is zero-padding followed by the application of a filter). Practically, speed and memory usage can be improved by combining the operations.

(Like LinFilt, Resample proved sufficiently useful to the rest of LDAS for its functionality to be abstracted into the Filters library.)

Listing 2.23: ResampleState class definition.

```

class ResampleState :
  public State
{
public:
  ResampleState(int p, int q, int n = 10, double beta = 5.0);
  ResampleState(const UDT& p, const UDT& q, const UDT& n =
    Scalar<int>(10), const UDT& beta = Scalar<double>(5.0));
  ResampleState(const ResampleState&);
  ResampleState(const UDT&);
  ResampleState(int p, int q, const Sequence<double>& b);
  virtual ~ResampleState() throw();
  ResampleState& operator=(const ResampleState&);
  int getP() const throw();
  int getQ() const throw();
  int getN() const throw();
  double getBeta() const throw();
  int getOrder() const throw();
  double getDelay() const throw();
  void getB(Sequence<double>& b) const;
  template<class T>
  void apply(valarray<T>& out, const valarray<T>& in);
  virtual ResampleState* Clone() const;
private:
  ResampleState();
  bool m_first;
  valarray<double> m_b;
  auto_ptr<Filters::ResampleBase> m_resample;
}; // class ResampleState

```


Usefully, the state of the resample action provides a method to extract the relative delay of the resampled sequence.

The criticisms of the previous classes apply equally to `Resample`, but it should be noted that `Resample` needlessly duplicates many of the accessors of `ResampleState`. This could charitably be viewed as the first steps to the adoption of a model removing the arbitrary division of a ‘stateful’ operation into two classes.

2.5 Actions

The simple data conditioning API command language, within the `–algorithms` option of the LDAS job, is C-like. It consists of a list of statements of the form

```
[identifier=]action([character-string(,character-string)*]);
```

i.e. an ‘action’ taking a parenthetic tuple of zero or more comma-separated character-strings, terminated with a semicolon, and with the result optionally assigned to a named identifier.

Listing 2.24: Valid data conditioning API commands.

```
| z = add(x, y);  
| i = length(z); |
```

This syntax is rigidly adhered to. Much of the ‘syntactic sugar’ of common languages is not supported, in the interests of keeping the syntax parser simple (and bug-free).

| Feature | Unsupported | Equivalent |
|--------------------|-------------------------------------|--|
| Trivial assignment | <code>y = x;</code> | <code>y = value(x);</code> |
| Infix operators | <code>z = x + y;</code> | <code>z = add(x, y);</code> |
| Nested functions | <code>w = add(x, sub(y, z));</code> | <code>t = sub(y, z); w = add(x, t);</code> |

Unlike C (but like C++), actions may be *overloaded* on the number and type of their arguments. This means that `z = add(x, y)` will perform one operation if `x` and `y` are scalars, and another if they are sequences.

The data conditioning API command language is interpreted, not compiled, and is not type safe. Any action can be called with any number and type of arguments, but it is a runtime error to do so for all but supported numbers and types of arguments.

Arguments are passed to actions as text, so it is not necessary that they be valid identifiers. For example, the `output` command interprets several of its arguments as field names for its XML-based output format.

The data conditioning API `-`algorithms are intended to form a pipeline. As such, there are *no* control flow statements (such as **if...else** branches or **while** loops); each statement is executed exactly once, in the order given.

The command language is parsed by the interpreted TCL layer; it is responsible for determining the number of arguments and the text associated with the return, action and argument identifiers. It then passes this information to the data conditioning API's library of compiled C++ code.

2.5.1 Call chain

A data conditioning API pipeline is represented in the library as a `CallChain` object. It stores both a series of objects corresponding to each action call, and a 'heap' of named UDT instances.

Listing 2.25: `CallChain` class definition.

```
class CallChain
{
public:
    void AppendCallFunction(string Function, vector<string> Params, string
        Return);
    void AddSymbol(string Name, UDT* Symbol); // overwrites if the
        symbol already exists
    UDT* GetSymbol(string Name); // throws exception if not found
    bool Execute();
    // ... (omitted)
};
```

The `AppendCallFunction` method is called by the TCL layer to pass on the identifiers associated with an action, its arguments and return data. Similarly the `AddSymbol` method is used to add the input data (under the names given in `-aliases`) to the `CallChain`.

Once all data and actions have been added to the `CallChain`, the `Execute` method runs the pipeline.

2.5.2 Call chain function

Just as a UDT type was required to enable the TCL layer and the call chain to store data without regard to its type, a base class `CallChain::Function` serves as the common interface for all signal processing actions.

Listing 2.26: `CallChain::Function` definition.

```
class CallChain
{
    // ... (omitted)
    class Function
    {
    public:
        Function(string Name);
        virtual ~Function();
        virtual void Eval(CallChain* Chain, vector<string>& Params, string
            Ret) const = 0;
        // ... (omitted)
    };
    // ... (omitted)
};
```

The `CallChain` class maintains a global list of named `Functions` so it can look up and invoke them from the name supplied in `AppendCallFunction`. The `Eval` method provides a common interface to the implementation of the action, supplying a pointer `Chain` to the calling environment, a vector `Params` of argument identifiers, and a Return value name (which is the empty string "" if no return value is given).

Specific actions are implemented by deriving a class from `CallChain::Function`; they implement the `Eval` operation to check and resolve the arguments, then forward them to a signal processing class.

2.5.3 mix

Class `MixFunction` inherits from `CallChain::Function` in a straightforward way to implement the mix action.

Listing 2.27: `MixFunction` class definition.

```
class MixFunction :
```

```

CallChain::Function
{
public:
  virtual void Eval(CallChain* Chain, vector<string> Params, string Ret)
    const; // override
  // ... (omitted)
};

```

It is the implementation of Eval where the interesting work occurs. The number of parameters is checked. Different numbers of parameters are handled by different code blocks. Each acquires a reference to the UDTs referred to by the symbols in the parameter list (throwing an exception if this is impossible). Once the references are obtained, the apply method is called on them. State is saved back to a symbol name if supplied, and finally the output is saved to the symbol name given as the action's return value. An unsupported number of parameters causes a BadArgumentCount exception to be thrown.

Listing 2.28: Mix action evaluator.

```

void MixFunction::Eval(CallChain* Chain, vector<string> Params, string
  Ret) const
{
  UDT* out = 0; // will store output
  switch (Params.size()) // how many arguments?
  {
  case 4: // y = mix(p, f, x, z); // this syntax
  case 3: // y = mix(p, f, x); // or this syntax
    { // use this implementation
      // -- get the arguments from the call chain -----
      UDT& phase(*(Chain->GetSymbol(Params[0]])); // GetSymbol
        throws an exception if symbol is not found
      UDT& frequency(*(Chain->GetSymbol(Params[1]]));
      UDT& in(*(Chain->GetSymbol(Params[2]]));
      // -- create and apply mixer using them -----
      Mixer mixer(MixerState(phase, frequency));
      mixer.apply(out, in);
      if (Params.size() == 4) // y = mix(p, f, x, z);
      { // there is a state to write to the chain
        Chain->AddSymbol(Params[3], new MixerState(mixer.getState()
          ));
      }
    }
  }
}

```

```

    }
  }
  break;
case 2: // y = mix(x, z); // this syntax
  { // uses this implementation
    // -- get the arguments from the call chain -----
    UDT& in(*(Chain->GetSymbol(Params[0]]));
    UDT& state(*(Chain->GetSymbol(Params[1]]));
    // -- create and apply mixer using them -----
    MixerState temp(state);
    Mixer mixer(temp);
    mixer.apply(out, in);
    // -- update the mixer state UDT -----
    Chain->AddSymbol(Params[1], new MixerState(mixer.getState()));
  }
  break;
default: // unsupported syntax
  throw BadArgumentCount(/* ... */);
}
// -- write result to call chain under name given by Ret -----
Chain->AddSymbol(Ret, out);
}

```

The implementations of `linfilt` and `resample` use similar techniques.

2.5.4 Simple actions

Operations already supported by C++ and its libraries need no programmer-level interface, but they do need to be exposed to the user. For example, the indispensable `slice` command can be simply implemented in terms of the `std::slice` STL class, within the body of a `CallChain::Function::Eval` override.

A difficulty arises again from resolving the exact type of the UDTs involved to apply the simple operations, for exactly the same reasons as the implementation of `apply(UDT)` is problematic. When the number of arguments and the number of types involved is large, the number of cases can be very large. This is unfortunately the case for basic and very important actions like `add`. Actual implementations often use generic programming and macros to automatically to relieve the programmer of some of this burden.

2.6 Testing

Testing is of such importance to the Data Conditioning API that contributors travelled to regular Mock Data Challenges (MDCs). A suite of automated tests mirrored the code as it was developed; an automated nightly build and check exercised the entire system and caught problems regularly and early.

For signal processing classes like Mixer, the test exercises all code paths through the class, and checks their self-consistency or consistency with known data (from, for example, MATLAB).

Typically comparisons are allowed to be approximate, as the precise order of operations employed will result in different values of floating point ‘noise’, the value of the least significant bits.

It is of course impossible to test every possible invocation of a method, but with knowledge of the internal structure of the method, it is possible to identify at least some corner cases and test for them as well as random cases hopefully representative of typical use.

A simple LDAS-wide `UnitTest` class is provided to help produce tests conforming to the expectations of the `make check` target, which allows the automatic testing of LDAS as part of its nightly build. The return value of the executable indicates success or failure of the test; the `UnitTest` class keeps track of this state.

Listing 2.29: Mixer class unit test.

```
#include "general/unittest.h"

General::UnitTest Test;

int main(int ArgC, char** ArgV) try
{

    Test.Init(ArgC, ArgV);

    try // simple sanity check
    {
        valarray<float> in(1.0, 122880); // a sequence of 122880 copies of
            1.0
        mixer(MixerState(0.0, -0.1)); // mix with -0.1 Nyquist frequency
        valarray<complex<float> > out; // sequence to store output
```

```

    mixer.apply(out, in); // apply the mixer
    for (valarray<float>::size_type i = 0; i < in.size(); ++i)
    { // compare each element of output with an independently computed
      value
      complex<float> expected(cos(-.1 * LDAS_PI * i), sin(-.1 *
        LDAS_PI * i));
      if (abs(out[i] - expected) > 1e-5)
      { // difference is too great
        Test.Message() << "Unacceptable error at " << i << endl;
        throw exception(); // abort test
      }
    }
    Test.Check(true) << "Sanity" << endl; // test passed
  }
  catch (...)
  {
    Test.Check(false) << "Sanity" << endl; // test failed
  }
  // more tests ...
  Test.Exit(); // returns number of failures
}
catch (exception& x)
{
  cout << "Exception: " << x.what() << endl;
  throw; // aborts (test suite fails)
}
catch (...)
{
  cout << "Exception: (non-standard type)" << endl;
  throw; // aborts (test suite fails)
}
}

```

In the case of testing the action wrapper for an operation, a call chain is manually prepared and executed. The tests have a different focus—correctness has already been ensured by the test of the implementation, so the testing concentrates on ensuring that in all possible cases the implementation is correctly invoked by the action.

Listing 2.30: Mix action (MixFunction class) unit test

```

// ... (initialisation)
// -- mix via call chain -----
CallChain commands;
Parameters arguments;
commands.AppendCallFunction(" double", arguments.set(1, "0.0"), " phase
");
commands.AppendCallFunction(" double", arguments.set(1, "-0.1"), "
frequency");
commands.AppendCallFunction(" double", arguments.set(1, "1.0"), " base")
;
commands.AppendCallFunction(" double", arguments.set(1, "122880"), " n"
);
commands.AppendCallFunction(" dvalarray", arguments.set(2, " base", " n"
, " x");
commands.AppendCallFunction(" mix", arguments.set(3, " phase", "
frequency", " x"), " y");
commands.Execute();
// -- mix directly -----
double phase = 0.; //
double frequency = -.1;
valarray<double> x(1.0, 122880);
Mixer mix(MixerState(phase, carrier));
valarray<complex<double> > y;
mix.apply(y, x);
// -- compare results -----
UDT* y_udt = commands.GetSymbol("y"); // extract named result from
call chain
Test.Check(y_udt) << "Sanity (output exists)" << endl;
Test.Check(UDT::IsA<Sequence<complex<double> >>(*y_udt)) << "
Sanity (output is a sequence)" << endl;
Sequence<complex<double> >& y_seq = UDT::Cast<Sequence<
complex<double> >>(*y_udt);
Test.Check(y_seq.size() == y.size()) << "Sanity (output size)" << endl;
for (valarray<complex<double> >::size_type i = 0; i < y.size(); ++i)
{
if (abs(y[i] - y_seq[i]) > 1e-5)

```



```

    { // difference is too great
      Test.Message() << "Unacceptable error at " << i << endl;
      throw exception(); // abort test
    }
  }
  // ... (further tests)

```

The above tests are automatically performed nightly and are designed so that any problem can be detected by machine and quickly brought to the attention of a maintainer.

The final stage in the testing of any piece functionality is to submit an actual LDAS job making sure the system works together as a whole:

Listing 2.31: LDAS mixing test job.

```

ldasJob { -name ***** -password ***** -email ***** } {
  conditionData
  -outputformat { ilwd ascii }
  -aliases { raw=mdc_input_data:chan_01:data }
  -algorithms {
    x0 = slice(raw,0,100,1); # Use 100 samples
    frequency = value(0.125);
    phase = value(0.1);
    z0 = mix(phi,frequency,x0);
    output(z0,...,z0,mixed whole);
    x1 = slice(x0,0,50,1); # Use first 50 samples
    phase = value(0.1); # Reset phase
    z1 = mix(phi,frequency,x1,state); # Mix, saving state
    output(z1,...,z1,mixed first half);
    x2 = slice(x0,50,50,1); # Use second 50 samples
    z2 = mix(x2,state); # Use state to continue mixing
    output(z2,...,z2,mixed second half);
  }
}

```

The above performs mixing on a sequence as a whole, then as two parts. A human will inspect the output and the test will pass only if the two methods yield identical results. These human-involving tests, and their capability to assess the ability of LDAS as a whole to perform a particular tasks, are the heart of MDCs.

2.7 Summary

LDAS is the core of LIGO's gravitational wave data analysis infrastructure. The data conditioning API component performs programmable signal processing as part of the data analysis pipeline. The author is a significant contributor to the data conditioning API. A suite of signal processing operations have been implemented in the API, and exposed to the user through a command language. Extensive testing of the components have been performed, both automatically and by hand. Hindsight may suggest better ways of implementing this functionality, but LDAS is a workable and extensively validated platform for gravitational wave data analysis.

Chapter 3

Line removal

High power in narrow frequency bands, *spectral lines*, are a feature of the output of interferometric gravitational wave detectors. Some lines are coherent between interferometers, in particular between the co-located 2 km and 4 km LIGO Hanford instruments. This is of particular concern to data analysis techniques, such as the stochastic background search, that use correlations between instruments to detect gravitational radiation. Several techniques of ‘line removal’ have been proposed. Where a line can be attributed to a measurable environmental disturbance, a simple linear model may be fitted to predict, and subsequently subtract away, that line. This technique has been implemented (as the command `oelsr`) in the LIGO Data Analysis System (LDAS). We demonstrate its application to LIGO S1 data.

We use data from a triple-coincidence epoch of the S1 science run in August–September 2002; all figures are composed of data drawn from the GPS times 714975000–714975600, except where noted. The LIGO interferometers have changed significantly since then: better shielding and equipment have reduced the magnitude of the lines we investigate, but the noise floor has also been reduced. The lines we investigate still remain prominent.

We also touch upon the use of line removal in the S1 stochastic background analysis. Though line removal was not employed to produce the upper limit on the strength of a cosmological stochastic background of gravitational waves [14], it was used to prove that spectral lines did not have a significant impact on the computed upper limit.

Some of the material in this chapter has appeared in Searle *et al.* [15].

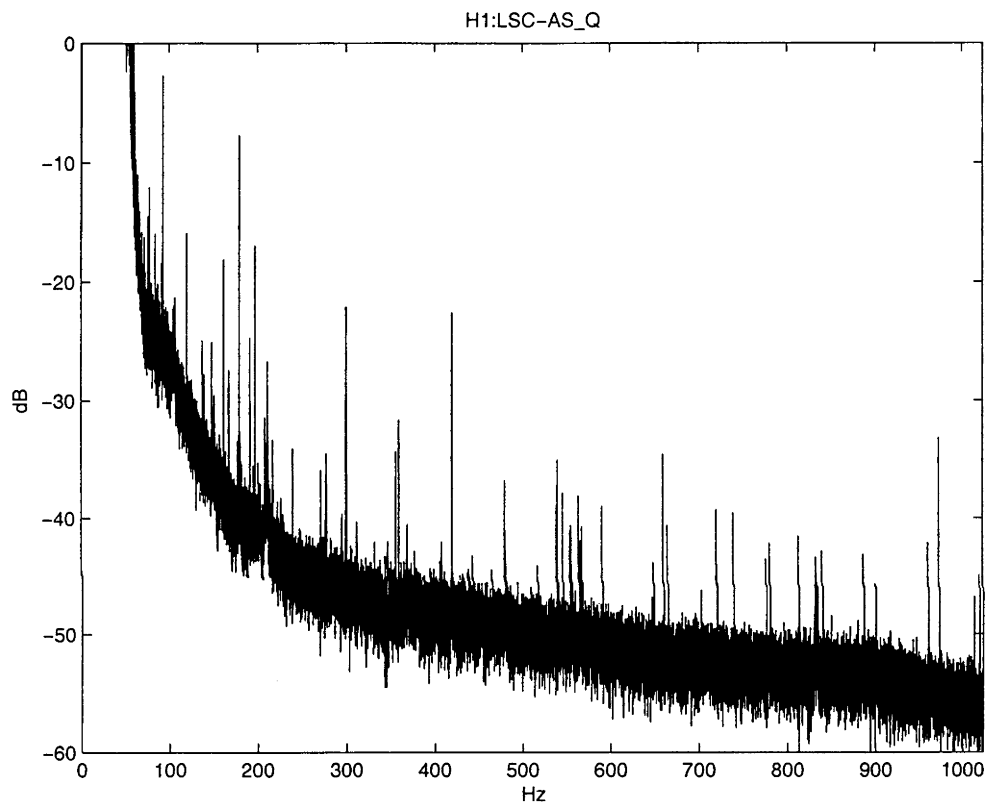


Figure 3.1: Lines in the power spectrum of the 4 km LIGO Hanford Observatory.

3.1 Motivation

The extreme sensitivity of the nascent LIGO instruments makes them particularly susceptible to contamination from the environment. Seismic noise, dictating the lower limit on the frequency of detectable astrophysical sources is perhaps the most dramatic example. Over the rest of the observation band, the theoretical limiting factors are thermal and shot noise. There are also, however, a number of narrow-band noise sources: spectral lines. Resonances of the mirror suspension wires (*violin modes*) are one example of these, but among the most prominent lines in each LIGO instrument are the combs of lines at 60 Hz and its harmonics (Figure 3.2), appearing across the entire observation band. These lines are the largest single factor in the coherence between the 2 km and 4 km LIGO Hanford interferometers (Figure 3.3).

Spectral lines are concerning for several reasons: they add power to the system; they increase the dynamic range of the data; they render the data non-gaussian. Depending on the nature of the search algorithms used, any of these effects can impair the detection rate, obscuring gravitational wave signals at or near the frequency of the lines. When the lines are correlated between interferometers, as is the case for some due to anthropogenic effects or large-scale environmental phenomena, they impair the ability of the interferometers to perform independent mutual verification.

A variety of approaches have been proposed to deal with spectral lines. The simplest is to ignore them; robust search algorithms have to be able to cope with non-gaussianity in all its forms, and we accept a potential decrease in sensitivity. The next simplest approach discards a frequency band containing the line, using a ‘notch’ filter; any attempt to extract information from the frequencies affected by the line is abandoned.

More sophisticated techniques attempt to construct a model of the line that can be subtracted from the data. These are referred to as *line removal* algorithms.

A Kalman filter proposes a linear model for the process producing the line, and fits that model to the history of the line to predict its future behaviour [16]. Where a line is one of a family of harmonics, Coherent Line Removal (CLR) exploits the commonality between the harmonics to approximate a shape shared by them all [17]. Once a model of the line is formed by either method, it is subtracted out of the raw data.

Both of the above methods use only a single time-series: the gravitational wave channel. This is undesirable, because the presence of a gravitational

wave signal has the potential to feed back into the behaviour of the line removal algorithm, possibly resulting in the attenuation of the gravitational wave signal as well as the line. For example, Sintès [17] finds that an injected sinusoid is only partially recovered by CLR; the sinusoid alters the common line shape approximation, not only resulting in its own partial removal but also reducing the quality of line removal at any the other harmonics where the shape is used.

In addition to the gravitational wave ‘output’ channel, gravitational wave observatories record thousands of channels of other data: some to control and monitor the instrument itself, others to detect environmental disturbances and enable the veto of candidate detections corresponding to physical anomalies. These other channels provide an additional source of information for line removal algorithms to use; and if a line removal algorithm can operate solely in terms of channels *not* sensitive to gravitational waves, it avoids the difficulty of altering the very signal it hopes to reveal. Allen, Hua and Ottewill [16] have comprehensively studied the removal of correlations between the gravitational wave channel and a number of environmental monitor channels, over all frequencies.

The comb of prominent lines at 60 Hz and its harmonics are due to the ubiquitous continental US power grid, which supplies alternating current at 60 Hz. The lines are strongly correlated with both the harmonics measured in the incoming electrical supply (Figure 3.4), and those observed by magnetometers distributed around the LIGO sites. Both of these environmental monitors are recorded in parallel with the interferometer output.

As spectral lines have the potential to impact a wide range of astrophysical searches, it is appropriate to implement line removal functionality as one of the tools available in the common LIGO data conditioning pipeline. We implement a time-domain spectral line removal algorithm as the data conditioning API action `oelslr` (Output Error Least Square Line Removal, pronounced ‘oelestra’). It permits line removal functionality to be straightforwardly added to the existing data conditioning API algorithms script for an astrophysical search, as another pre-processing stage in the analysis ‘pipeline’. Such a line removal stage has already been added to the data conditioning script for the stochastic background search.

Our implementation is intended as the first step in adding line removal functionality to the data conditioning API. Similarly to Allen *et al.* [16], it makes use of the additional information available in the form of a measurement of the disturbance to the instrument, in particular the voltage monitor

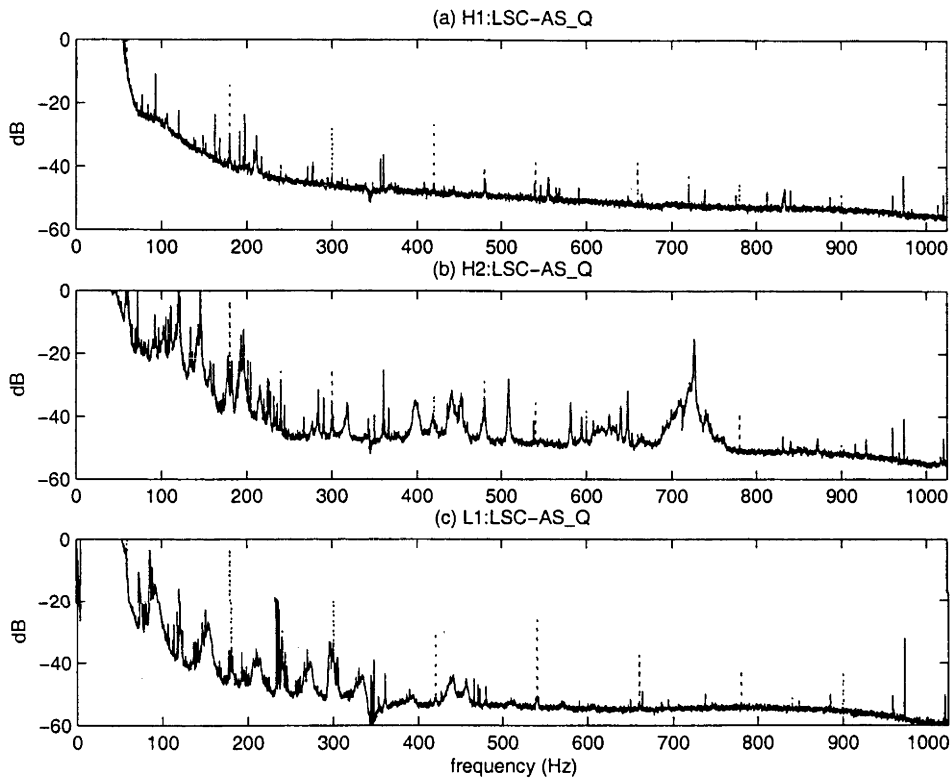


Figure 3.2: LHO (a) 4 km and (b) 2 km interferometers and (c) LLO 4 km interferometer output power spectra (uncalibrated), before (dotted) and after (solid) line removal.

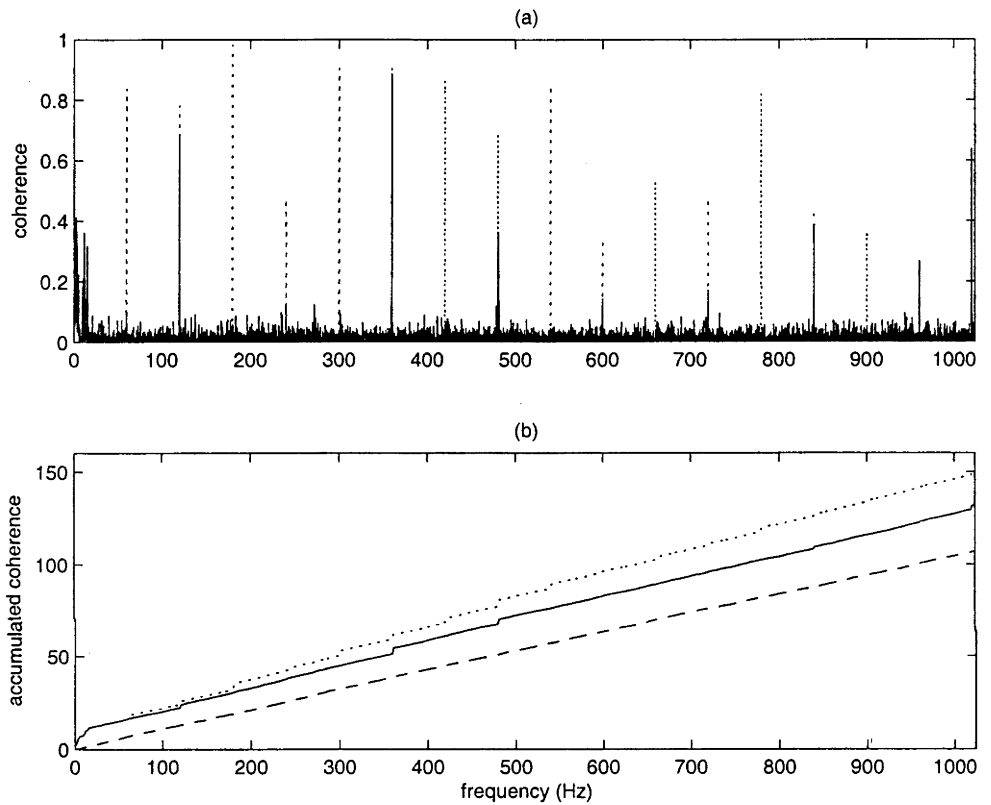


Figure 3.3: (a) Coherence of, and (b) accumulated coherence of, H1:LSC-AS_Q and H2:LSC-AS_Q before (dotted) and after (solid) line removal, with the accumulated coherence of H1:LSC-AS_Q and L1:LSC-AS_Q (dashed) provided for reference.

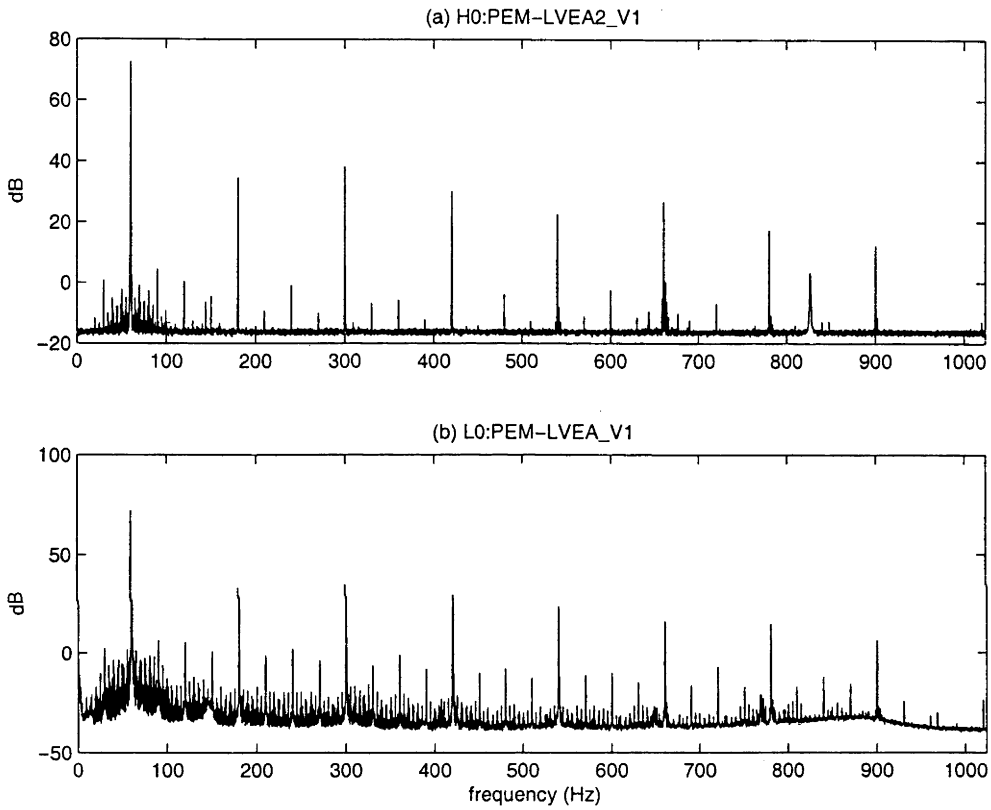


Figure 3.4: (a) LHO and (b) LLO voltage monitor channel (uncalibrated) power spectra.

channels recorded by an observatory (Figure 3.4). Unlike Allen, we implement the algorithm in the time domain, to facilitate easy continuation of the algorithm over LDAS jobs, as for most other data conditioning API actions. With a known, measured (albeit imperfectly) disturbance, we may use system identification theory techniques [18] to construct a model of the interferometer response to the disturbance from actual data recorded over some epoch. At subsequent times, the model can then be used to predict the interferometer’s response from the measured disturbance; this prediction can then be subtracted away to remove those lines from the interferometer output.

3.2 Design

The line remover model proposes that a spectral line in a discrete time series $y[t]$ with no white noise present is due to a measurable disturbance $u[t]$. It estimates a *model* M such that

$$\hat{y} = M(u) \tag{3.1}$$

predicts the spectral line. Any component of $y[t]$ predictable (by this type of model) from $u[t]$, such as the spectral line, should thus be *removed* from the time series $r[t] = (y - \hat{y})[t]$. Features that cannot be predicted from $u[t]$ should be unaltered.

The Finite Impulse Response (FIR) model is the simplest special case of a *regression model*¹, where the (possibly complex) input u and output y of a system are related by (following Ljung [18], where t is a discrete sample number)

$$y(t) \approx b_1 u(t-1) + b_2 u(t-2) + \dots + b_m u(t-m), b_1 \dots b_m \in \mathbb{C}, \tag{3.2}$$

where the model for the system is $\theta = [b_1 \dots b_m]^T$. The model order m controls the range of samples contributing to the model output.

If we define $\varphi(t) = [u(t-1) \dots u(t-m)]^T$, then $\hat{y}(t|\theta) = \varphi^T(t)\theta$ is the modelled output of the system. For given y and u , the best model $\hat{\theta}_N$ (which minimises, over θ , the sum over N samples of the square of the prediction

¹Line removal in the datacond API was originally designed to use the Output-Error (OE) model, but the complexity of an OE model estimator has delayed its implementation.

error) is given by

$$\hat{\theta}_N = \arg \min_{\theta} \sum_{t=1}^N |y(t) - \varphi^T(t)\theta|^2, \quad (3.3)$$

and can be determined analytically (where $\overline{\varphi(t)}$ denotes the complex conjugate of $\varphi(t)$):

$$\hat{\theta}_N = \left[\sum_t^N \overline{\varphi(t)}\varphi^T(t) \right]^{-1} \sum_t^N y(t)\varphi(t). \quad (3.4)$$

For an FIR model with a white noise (or *error*) term, $e(t)$, such that

$$y(t) = b_1u(t-1) + b_2u(t-2) + \dots + b_mu(t-m) + e(t), \quad (3.5)$$

it is important to note that $\hat{\theta}_N$ is an unbiased estimate of θ , converging as $N^{-\frac{1}{2}}$ [18].

In the context of gravitational wave data analysis, we assume that the detector output $y(t)$ would consist of white noise $e(t)$, but for another channel $u(t)$ that linearly and additively contaminates it with $b_1u(t-1) + b_2u(t-2) + \dots + b_mu(t-m)$. To remove the contamination, we estimate b_i and subtract away the model prediction from the measured data:

$$y_r(t) = y(t) - \varphi^T(t)\hat{\theta}_N. \quad (3.6)$$

Where the contamination is localised in frequency space, as for spectral lines, it would be wasteful to apply the method to the raw time series. (It would also introduce problems of the kind encountered in §3.3.1.) Using pre-existing components of the data conditioning API we can produce new time series of fewer elements containing only the information from a particular narrow frequency band.

If $u(t)$ is sampled with Nyquist frequency f_{Ny} and a line is restricted to $f \pm f_{Ny}/n$ for some integer n , then u is first mixed down to zero frequency with multiplication by $e^{-i2\pi ft/f_{Ny}}$. The non-trivial data conditioning API `resample` algorithm [19] is then used to down-sample (by a factor of n) to a new series with Nyquist frequency f_{Ny}/n (including a filtering stage to prevent the aliasing of high-frequency components into the result). Restricting the bandwidth to an integer fraction of the Nyquist frequency allows the time-domain resampling to be performed simply and efficiently. The

down-sampling factor n may typically be quite large—we use 128 below—to identify a narrow band. A side-effect of the resampling algorithm is that any sequences processed by the line remover must contain an integral multiple of n samples, making it advantageous to make n correspond to some common divisor of the desired sample counts—typically a power of 2. It is on this pre-processed version of u that the model is fitted. The process is reversed to produce \hat{y} by up-sampling (again including a smoothing filter) and up-mixing the model prediction. These operations have been abstracted into the `datacondAPI::BandSelector` class, and have already seen reuse in the data conditioning API implementation of the Kalman filter.

In the data conditioning API, line removal is performed using the `oelslr` action to both estimate θ and predict \hat{y} . (Recall from §2.5 that, like its underlying C++ implementation [2], the `datacondAPI` command language allows *overloading* of function names, so that `oelslr` may perform different tasks depending on the number and type of arguments supplied to it.)

$$\theta = \text{oelslr}(y, u, \frac{f}{f_{Ny}}, n, m); \quad (3.7)$$

$$\hat{y} = \text{oelslr}(u, \theta); \quad (3.8)$$

The general purpose `sub` action is used to perform the final subtraction.

$$y_r = \text{sub}(y, \hat{y}); \quad (3.9)$$

The time-domain causal linear filters employed both in the band-selection and the model implementation introduce transients and time-delays into the prediction, which manifest in the first prediction from any model (subsequent calls are not affected as the linear filters are preserved in the model internal state θ). The model implementation as a causal linear filter produces start-up transients invalidating the first nm samples of the prediction. The band-selection also truncates the prediction by an implementation-defined multiple of n samples (128 n samples for the current `resample` implementation). These issues can be simply addressed by providing u for $[t_1 - \delta t, t_2 + \delta t]$ where \hat{y} for $[t_1, t_2]$ is required; for typical parameters, $\delta t = O(\text{seconds})$.

A line removal LDAS ‘job’ is composed as follows. An estimation era is sliced from incoming channels (aliased as u and y).

```
|
|     ue = slice(u, 0, 1228800, 1);
|     ye = slice(y, 0, 1228800, 1);
|
```

Recalling the definition of slice from 2.5, we have sliced off 1228800 samples, starting from 0, with stride 1. For these 2048 Hz channels, this corresponds to the first 10 minutes of each. To remove the 180 Hz line, we estimate a model θ from y_e and u_e , in a region of $(0.17578125 \pm 128^{-1})f_{Ny}$, with order of 8.

```
|         theta = oelsr(ye, ue, 0.17578125, 128, 8);         |
```

The next stage is to slice a prediction era from u . We wish to predict y for the subsequent 10 minutes, and allowing (for simplicity) a generous δt of 1 minute, we use a slice of u extending for 12 minutes from 9 minutes after the beginning of the channel.

```
|         up = slice(u, 1105920, 1474560, 1);         |
```

To produce a prediction yp requires only the model θ and the predictor up .

```
|         yp = oelsr(up, theta);         |
|         yp = slice(yp, 122880, 1228800, 1);         |
```

Once the prediction has been produced, we reset it to a slice of its own middle 10 minutes (beginning 1 minute into the almost 12 minute raw prediction), effectively discarding the start-up transients and trailing truncation.

```
|         ym = slice(y, 1228800, 1228800, 1);         |
|         yr = sub(ym, yp);         |
```

We store the measured values of y for the corresponding times in ym , in preparation for subtracting the prediction from the measurement to produce the line-removed sequence yr . The sequence yr can then be output to other LDAS APIs for further processing by astrophysical searches, or, as in §3.3, written to file for inspection.

3.3 Characterisation

The obvious figure of merit for line removal is the change it produces in the sensitivity of the astrophysical searches whose data it pre-processes. Such tests have been already performed by the author in the specific case of the stochastic background search [14], conclusively demonstrating the robustness of the search algorithm in the presence of strong spectral lines (§3.4). Here

we demonstrate reductions in line power and line coherence, two effects can reasonably be expected to impact many astrophysical searches.

Though line removal is conceptually simple, the quantification of line removal quality is not a trivial matter. The obvious metric is the remaining power in the line. Yet by this metric the notch filter—or even a zero multiplier—are optimal methods. The line *should* be removed to the noise floor, but this requires the identification of what the noise floor is under and around the line. The frequency band around the line should be unaltered, but in practice some small alteration will occur, requiring some judgement of the relative merits of alterations to the line and nearby frequencies. Gaussianity should be improved, but Sintès [17] demonstrates inconclusive results by this metric, as spectral lines were not the primary source of non-Gaussianity.

This introduces problems with automated testing and verification procedures. Ultimately, the only metric that makes sense is the ability of a line removal technique to improve the sensitivity of a gravitational wave search. In this section we assess performance in recovering an injected sinusoid masked by the line, and in reducing coherence between two observatories. The first test is equivalent to the tests carried out in [17].

It would be desirable to be able to conduct a direct comparison with other algorithms such as [17, 16], some of which are publicly available. Existing publications on these codes, however, use much less modern data from prototype interferometers. To produce a meaningful comparison would require the author to run the codes on the same modern data segments as OELSLR. As these codes are not part of LDAS, however, this is not a simple matter—new, non-LDAS pipelines would have to be constructed from scratch, or alternatively, the codes themselves would have to be retrofitted into LDAS. The limited success of a similar project by an earlier member of our group [20] convinced us that the difficulty of such an undertaking was prohibitive under present circumstances.

3.3.1 Injection

Three sinusoids of equal amplitude were added to the H1:LSC-AS_Q channel, at approximately 299.4, 300.0 and 300.6 Hz. Their amplitude was selected to be intermediate between the line amplitude and the noise floor.

Ideally, we expect to see all traces of the 300 Hz line removed to the level of the surrounding line, leaving the surrounding features unaltered, and the recovery of the injected line.

We wish to determine the robustness of the system and investigate the quality of line removal achieved by a wide variety of parameters.

Model estimation depends on four parameters: the central frequency, the bandwidth, the model order, and the length of the estimation epoch. It is impractical to cover this four-dimensional parameter space, so we vary each parameter separately from a set known anecdotally to produce acceptable results. The parameters were 300 ± 8 Hz with order 8, with estimation era GPS 714974400–714975000 and line removal era GPS 714975000–714975600.

Note in particular the characteristics produced by this reference case (Figure 3.5). The line itself is removed down to the level of the noise floor, and the noise floor itself is unperturbed, at least by any effect of comparable magnitude to it. The injected signals are all present at their full amplitudes; the signals on either side of the line are in fact unperturbed. The only undesirable feature is the broadening of the signal recovered from beneath the line.

Examination of the figure shows the difficulties in forming a simple test for the effectiveness of the line remover. It should test that the line is diminished, but this requires an automated identification of just where the line is, and a judgement as to just where its flanks taper into insignificance. It should test that the line is removed to the noise floor, but this requires the identification of the noise floor, which in turn requires the identification of any features that are not representative of the noise floor. The result should not deviate significantly from the original, except at the location of the line. The problems are effectively ones of pattern recognition, where the human eye and brain excels, and algorithms do quite poorly; any algorithm would be complicated, replete with ‘magic numbers’ and pragmatically justified heuristics. Relying on such techniques would scarcely add confidence to the analysis. We resort to visual inspection to detect artifacts, but employ specific measures of quality where appropriate (while recognising they cannot tell the whole story). The ultimate such measure is of course the effect on a gravitational wave search.

Central frequency

The frequency of the alternating current electrical supply—and thus the harmonics it induces in the gravitational wave detector—are allowed to wander by some small fraction of a Hertz by the generating utilities (this assists in load balancing across the electrical grid). Other lines may wander; many

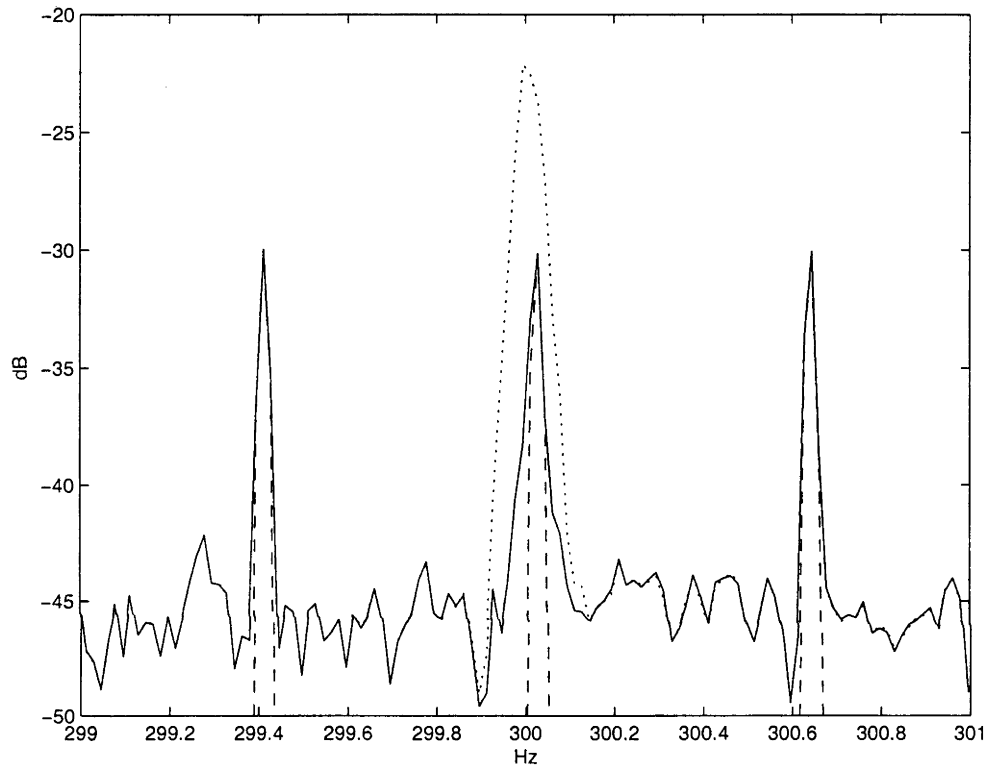


Figure 3.5: Power spectrum of H1:LSC-AS_Q and added sinusoids before (dotted) and after (solid) application of line removal to 300 ± 8 Hz with order 8. The power spectrum of the injected sinusoids alone is dashed. The estimation era was GPS 714974400–714975000 and the line removal era was GPS 714975000–714975600.

may have no precisely known frequency. It is important to characterise the performance of the line removal code when the line is not centrally located. Within the majority of the interval of removal, there is no reason to expect any effect, but at the edges of the interval, the filters employed in down- and up-sampling should produce a fall-off in the accuracy and amplitude of the prediction, and a corresponding decrease in effectiveness.

In the $f = 300\text{--}307$ Hz regime there is no change in the effectiveness of the line removal; the results are substantially as in Figure 3.5

At $f = 308$ Hz, where the edge of the interval corresponds to the line itself, the line removal algorithm has only a slight effect: the line is somewhat diminished, as seen in Figure 3.6.

For $f = 309$ Hz the line lies outside the nominal affected area. No effect from line removal is visible around the line, Figure 3.7. Note that around f , there is a minor perturbation to the noise floor, visible in Figure 3.8. The model *should* consist of all zeros in this case, but in practise the finite data on which the correlations are taken introduce uncertainties from random correlations that fail to average out completely; the result is a weak transfer of noise from the predictor channel to the line removal channel. This is unfortunate, but turns out not to be a cause for concern. It will be addressed in §3.3.2.

Bandwidth

The bandwidth of the line remover is another property that should ideally have comparatively little effect. However, the bandwidth affects the number of terms considered in the determination of the model, and if the model order is large and the bandwidth small the model may suffer from high variance. Correspondingly, if the model order is small and the bandwidth is large, the line may be a relatively trivial component in the whole, and the optimal solution will be dominated by the minimisation of chance correlations in the noise rather than the removal of the line. Since these chance correlations will not persist at subsequent times, what cancelled them in a previous epoch instead becomes an additive noise source of comparable magnitude to the noise floor itself.

As the bandwidth is decreased (by increasing the down-sampling ratio) no effect is observed up to the maximum possible down-sampling ratio of 4096. (This is the greatest power of two by which the sequence length is divisible.)

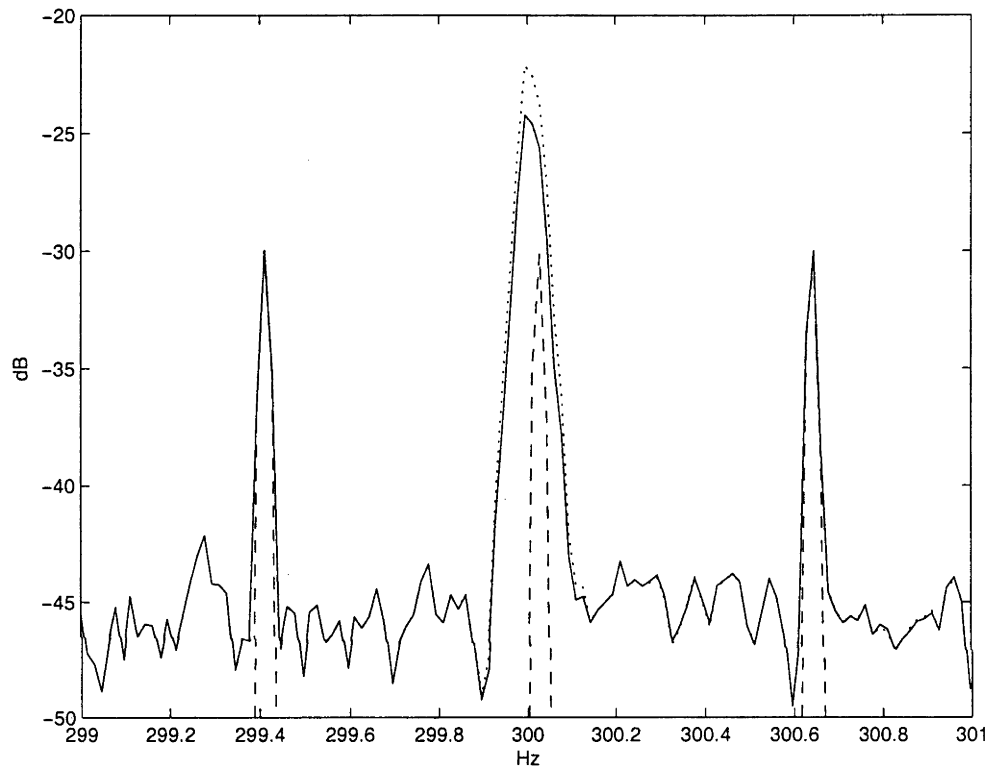


Figure 3.6: Power spectrum of H1:LSC-AS_Q and added sinusoids (dashed) before (dotted) and after (solid) application of line removal to 308 ± 8 Hz with order 8. The estimation era was GPS 714974400–714975000 and the line removal era was GPS 714975000–714975600. The line lies on the edge of the removal interval, and is only slightly attenuated. Other frequencies are unaffected.

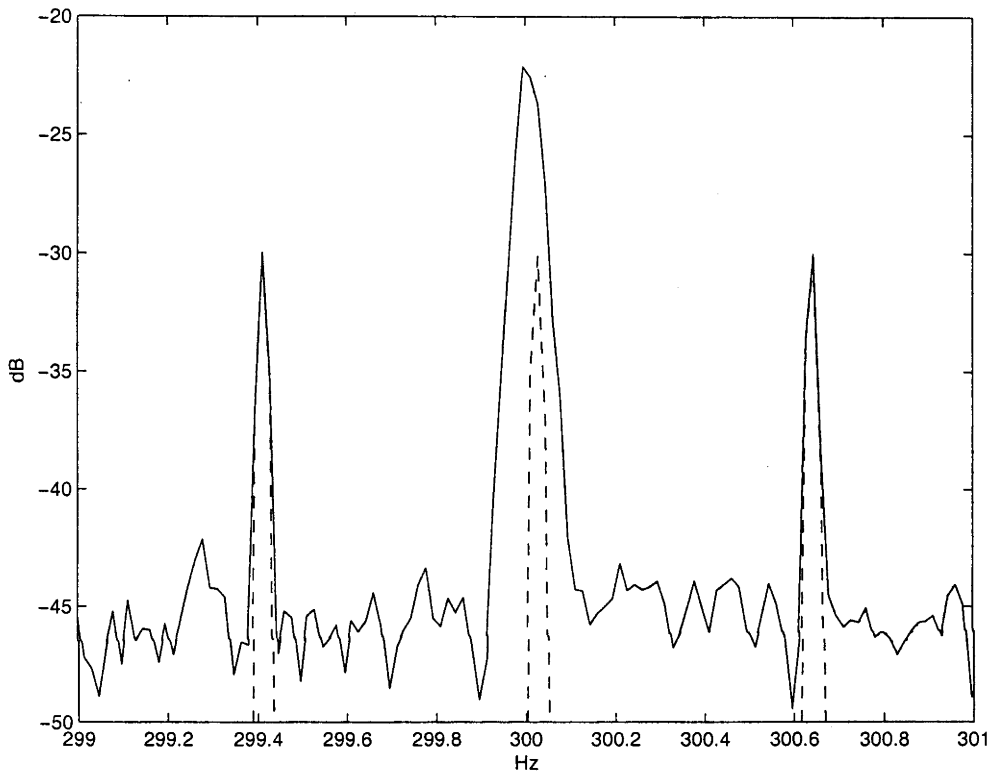


Figure 3.7: Power spectrum of H1:LSC-AS_Q and added sinusoids (dashed) before (dotted) and after (solid) application of line removal to 309 ± 8 Hz with order 8. The estimation era was GPS 714974400–714975000 and the line removal era was GPS 714975000–714975600. The line lies outside the removal interval. No frequencies are significantly affected.

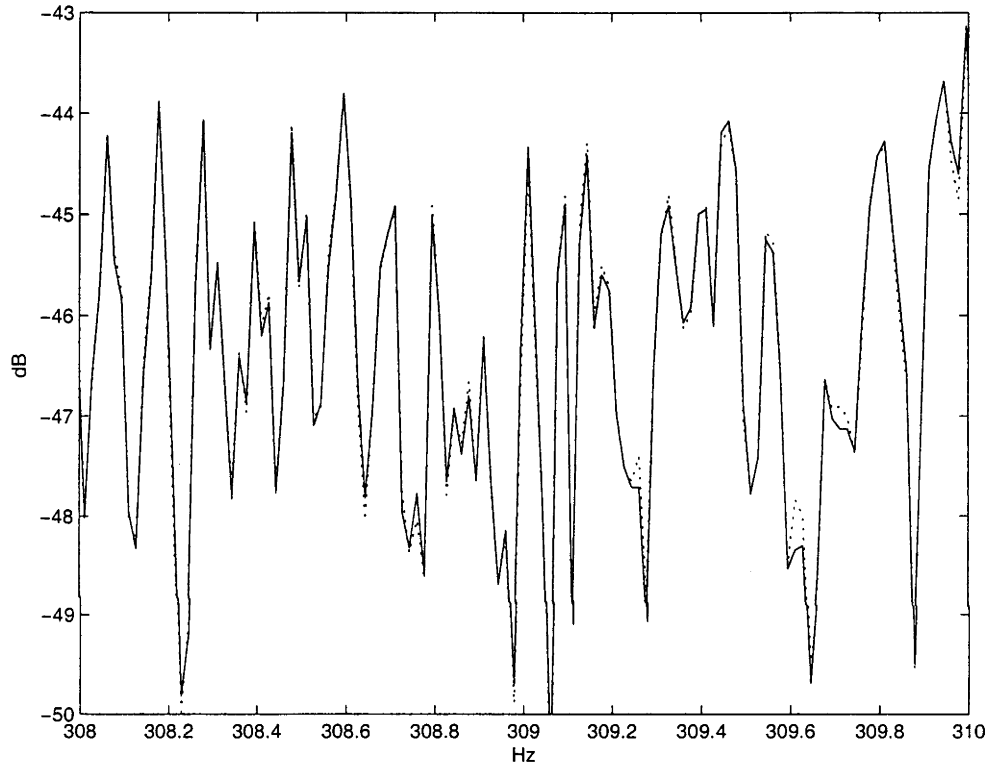


Figure 3.8: Power spectrum of H1:LSC-AS_Q before (dotted) and after (solid) application of line removal to 309 ± 8 Hz with order 8. The estimation era was GPS 714974400–714975000 and the line removal era was GPS 714975000–714975600.

As the bandwidth is increased (by decreasing the down-sampling ratio), the line removal becomes less effective, and broadband noise is introduced. With a down-sampling ratio of 16, the line is partially removed (Figure 3.9) but the model has added noise to the surrounding band (Figure 3.10).

Model order

The model order is intimately bound up with the bandwidth, and the effects posited above can also be induced by variation of the model order. The determination of a model order able to physically model the process causing the line is a separate issue. If the process is simple over the width of the line, then we might expect that a trivial model—an amplitude adjustment and phase delay, provided by multiplication by a complex scalar—would be sufficient. If not, a higher-order model will be needed. (One possible cause would be additional noise in the prediction channel; a higher-order model would be able to average some of this noise out of its prediction by relying on the contributions of multiple samples.)

As the model order decreases, there is little effect on the quality of the prediction. The trivial model—multiplication by a single complex scalar—produces excellent results (Figure 3.11).

On the other hand, as the model order increases the recovery of the signal is unaffected. The method begins to perturb the noise floor, however, as seen in Figure 3.12.

Estimation era

As the length of the estimation epoch decreases, so does the number of data points and hence the variance of the model estimate increases. Signs of deterioration are evident at 60 seconds; the quality of the removal of the line using only 4 seconds is reasonable, but the perturbation to the noise floor is obvious (see Figure 3.13).

No improvements were witnessed at shorter times, implying that the timescale of variation of the physical process is longer than 10 minutes. Since the data conditioning API handles data on that timescale, all is well in this regard.

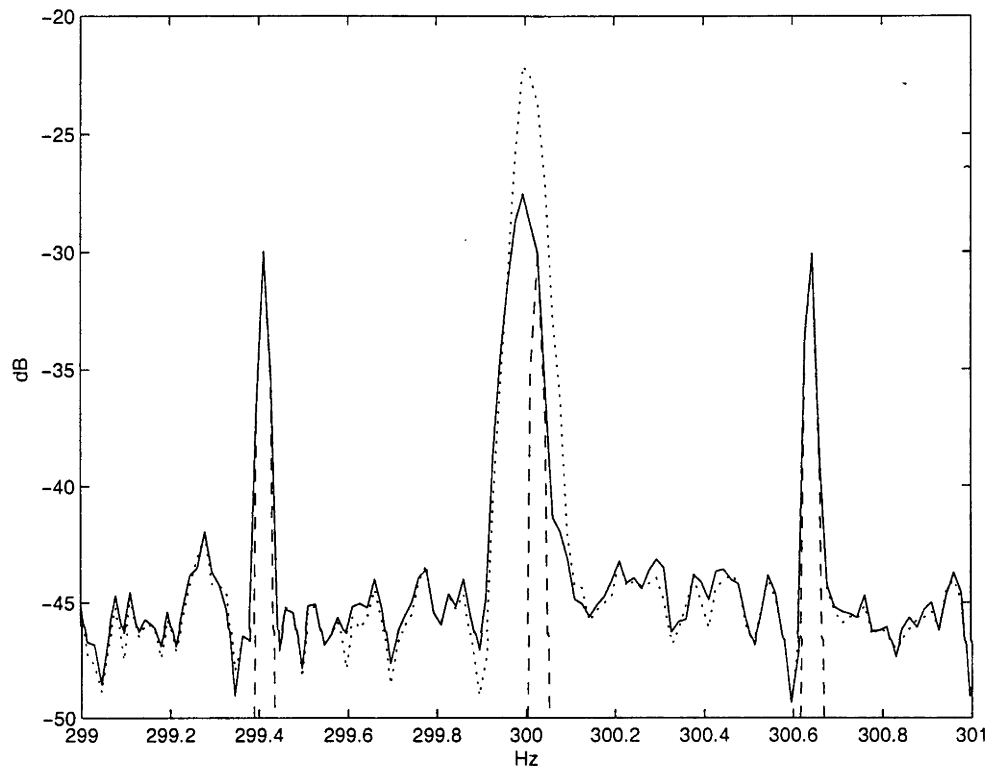


Figure 3.9: Power spectrum of H1:LSC-AS_Q before (dotted) and after (solid) application of line removal to 300 ± 64 Hz with order 8. The estimation era was GPS 714974400–714975000 and the line removal era was GPS 714975000–714975600. The line is partially removed; note the noise floor has been perturbed.

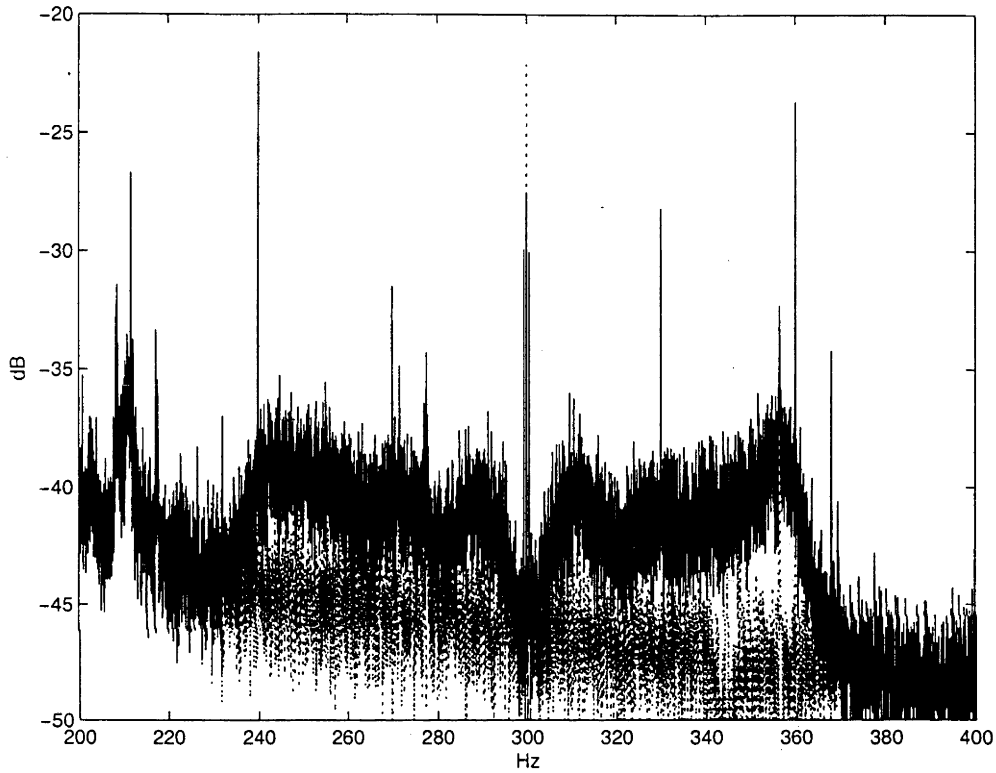


Figure 3.10: Power spectrum of H1:LSC-AS-Q before (dotted) and after (solid) application of line removal to 300 ± 64 Hz with order 8. The estimation era was GPS 714974400–714975000 and the line removal era was GPS 714975000–714975600. The algorithm has introduced noise across its band of operation.

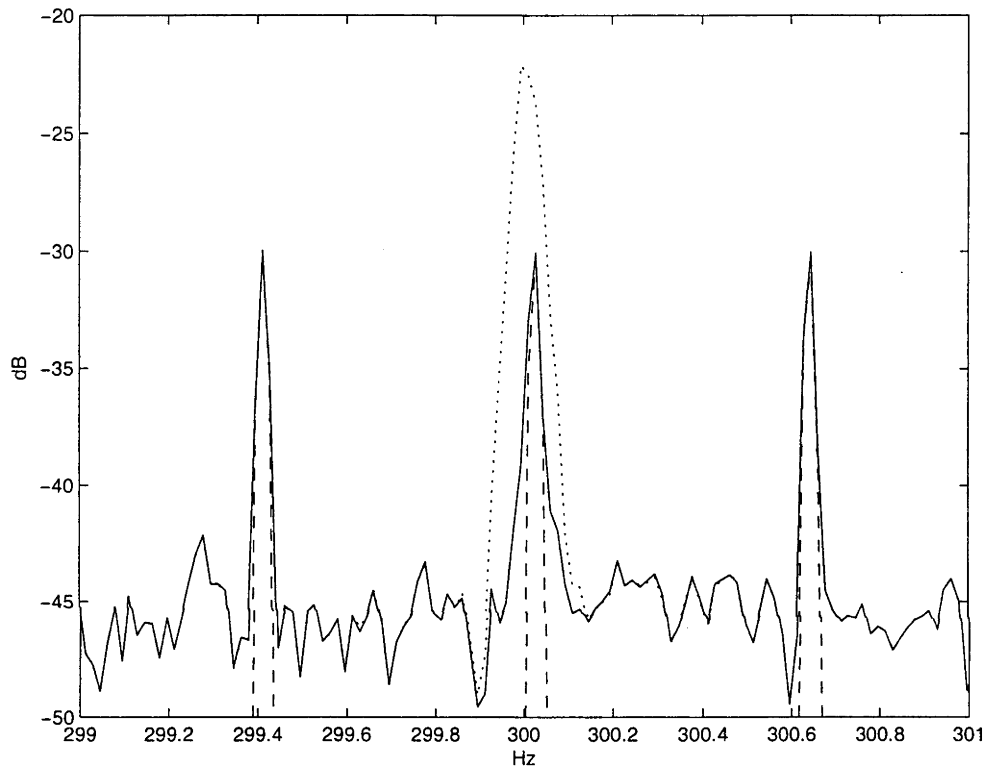


Figure 3.11: Power spectrum of H1:LSC-AS_Q before (dotted) and after (solid) application of line removal to 300 ± 8 Hz with order 1. The estimation era was GPS 714974400–714975000 and the line removal era was GPS 714975000–714975600. The line has been removed to the noise floor; there is little evidence of broadening of the signal.

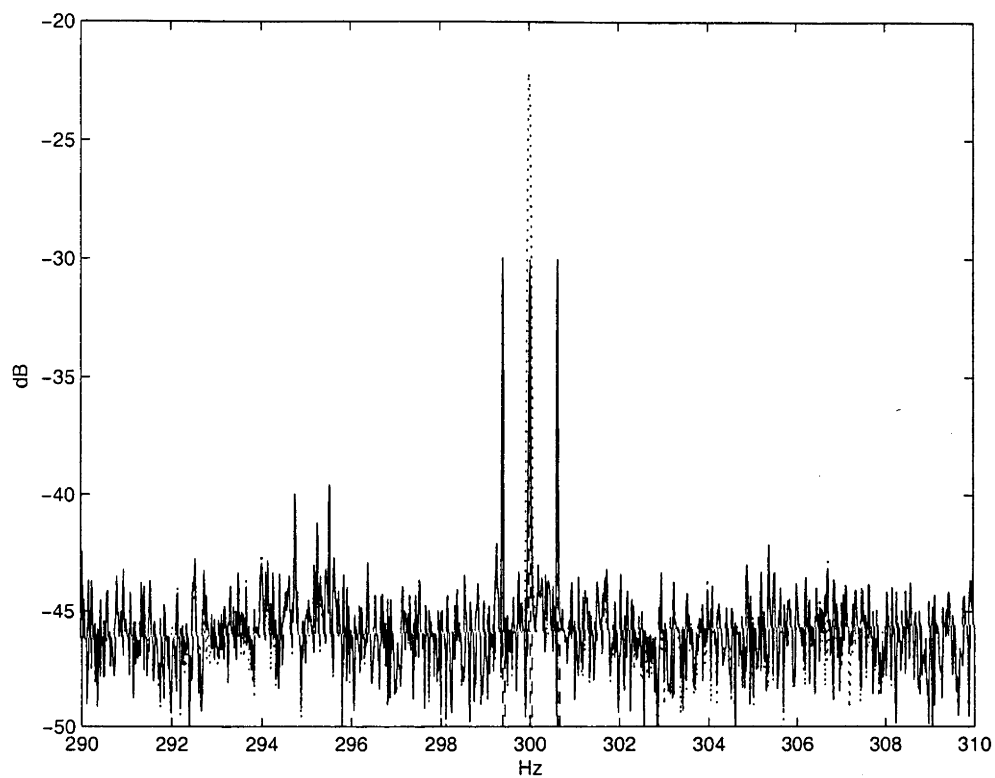


Figure 3.12: Power spectrum of H1:LSC-AS_Q before (dotted) and after (solid) application of line removal to 300 ± 8 Hz with order 128. The estimation era was GPS 714974400–714975000 and the line removal era was GPS 714975000–714975600. Note perturbation of the noise floor throughout the line removal band.

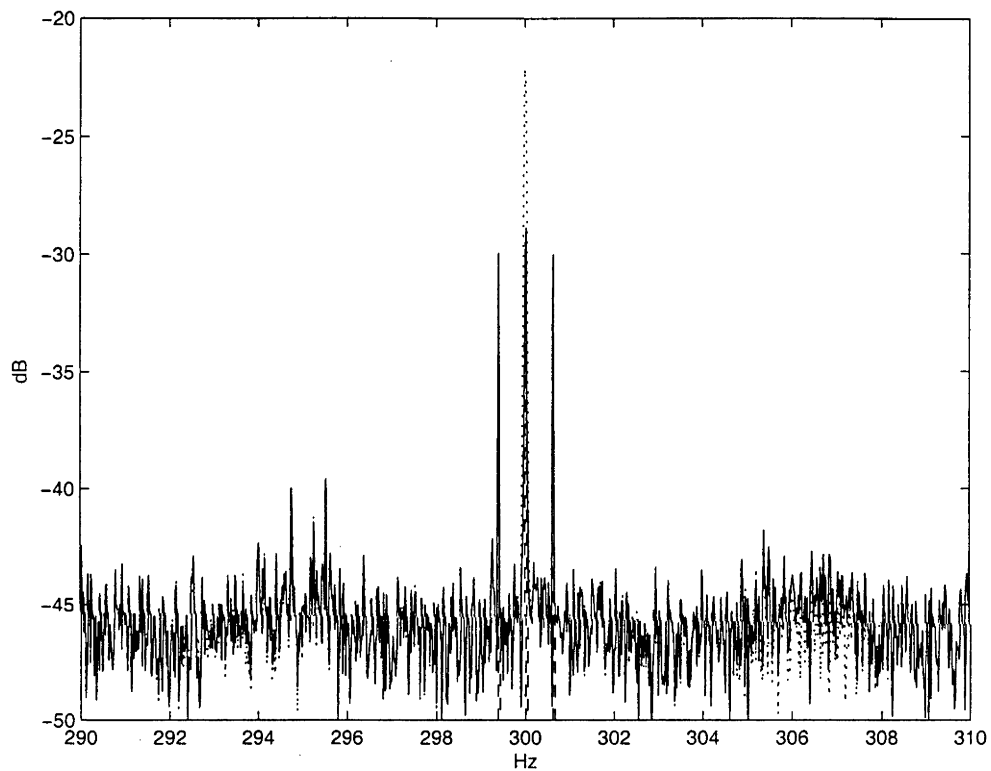


Figure 3.13: Power spectrum of H1:LSC-AS_Q before (dotted) and after (solid) application of line removal to 300 ± 8 Hz with order 8. The estimation era was GPS 714974996–714975000 and the line removal era was GPS 714975000–714975600. Note perturbation of the noise floor throughout the line removal band.

3.3.2 Coherence

The intent of this section is to use a quantification inspired by the stochastic background search.

Data taken during LIGO runs is formatted as *frames* of named channels. The 4- and 2-km Hanford and 4-km Livingston observatories ‘gravitational wave channels’ are, respectively, H1:LSC-AS_Q, H2:LSC-AS_Q and L1:LSC-AS_Q. Here we use data from the S1 Science Run, specifically from a stretch of ‘triple-coincidence’ data from GPS 714974400–714975660, during which time all interferometers were locked; all figures are produced using the 10 minutes of data from GPS 714975000–714975600.

The LSC-AS_Q channels of all three interferometers show lines at multiples of 60 Hz in their power spectra (Figure 3.2), particularly for odd harmonics (60 Hz, 180 Hz, 300 Hz, ...). These lines are strongly coherent between H1 and H2 (Figure 3.3), but not between H1 and L1 or H2 and L1 (not shown). The lines are attributed to interference from the 60 Hz alternating current mains supply, and the coherence is attributed to the fact that a common mains supply is shared between H1 and H2 at Hanford, WA, but not by L1 at Livingston, LA, as the electrical grid is not coherent between the sites on this short timescale [21].

The power spectra of the mains voltages at both observatories (Figure 3.4) exhibit prominent lines at the odd harmonics of 60 Hz, and weaker lines at the even harmonics. At each observatory, the voltage of the incoming mains supply is measured by several monitors and recorded to corresponding channels; we use H0:PEM-LVEA2_V1 and L0:PEM-LVEA_V1. The LSC-AS_Q channels are typically strongly coherent with their local voltage monitors at odd harmonics, and weakly coherent at even harmonics (Figure 3.14). This indicates that the voltage monitor channels should be good predictors of the odd harmonics, and fair predictors of the even harmonics, given the simple linear model used by the line remover.

Assuming that we may regress *:LSC-AS_Q against *0:PEM-LVEA*_V1, we construct an LDAS job to separately remove lines at each of the 17 multiples of 60 Hz beneath the 1024 Hz Nyquist frequency of the 2048 Hz *0:PEM-LVEA*_V1 channels (the *:LSC-AS_Q channels are downsampled from 16384 Hz to 2048 Hz before this stage of the data conditioning). This proceeds for each line as in Appendix A, with the exception that for each channel we store a single prediction sequence consisting of the sum of the predictions for each of the lines for that channel—a single unified \hat{y} predicting all lines.

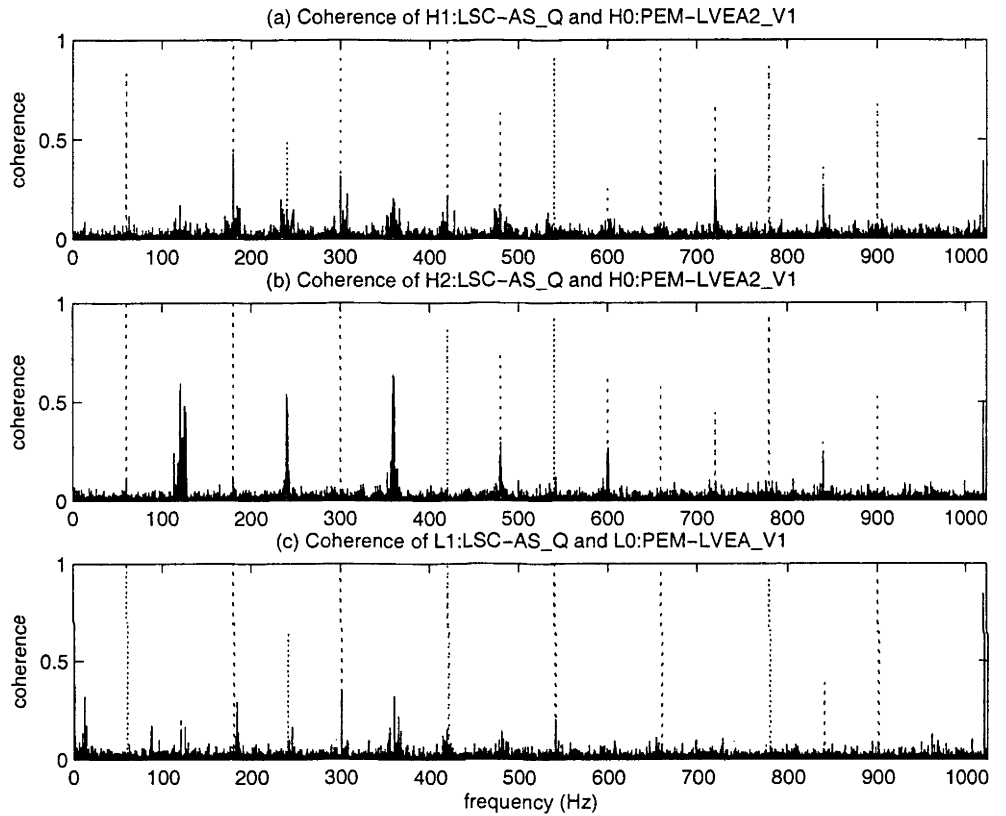


Figure 3.14: Coherence of (a) H1:LSC-AS_Q, (b) H2:LSC-AS_Q and (c) L1:LSC-AS_Q with their respective voltage monitor channels, H0:PEM-LVEA2_V1 and L0:PEM-LVEA_V1, before (dotted) and after (solid) application of the line removal technique described in §3.3.

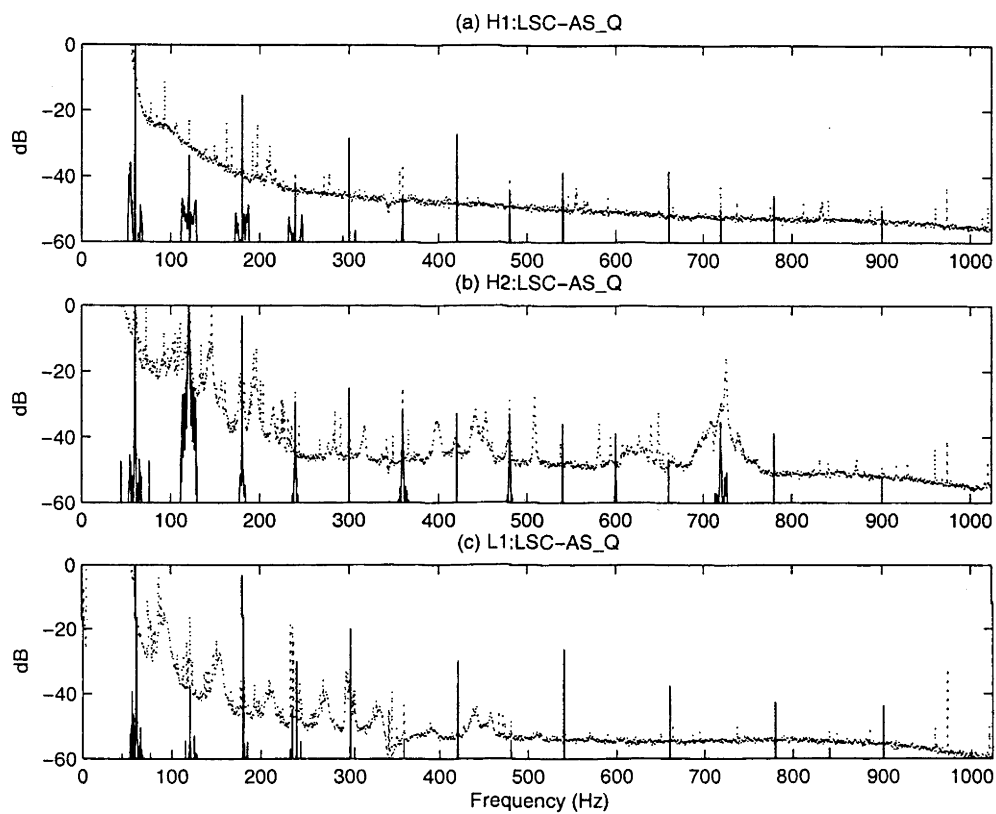


Figure 3.15: Power spectra of the prediction for (a) H1:LSC-AS_Q, (b) H2:LSC-AS_Q and (c) L1:LSC-AS_Q (solid). Corresponding power spectra of the channels are provided for reference (dotted).

Several features are notable on the power spectra of the predictions (Figure 3.15). First, the power of the prediction never exceeds the power of the LSC-AS_Q channel. Outside the bands selected for line removal, the power is 5–10 orders of magnitude below the *:LSC-AS_Q noise floor. Within the bands selected for line removal, the power is at least 2 orders of magnitude below the *:LSC-AS_Q noise floor. Most importantly, for the lines themselves, the power of the prediction is comparable to the power of the lines.

When the prediction is subtracted from the measured channel, the lines, as measured in the power spectra of Figure 3.2, are affected to varying degrees. Many are no longer visible above the noise floor; others have been reduced but are still present; some are unaffected. The residual coherence (Figure 3.14) between the line removed channels and their predictors is similar. For most lines, the coherence has been reduced; for many there is no residual coherence above the noise level of the estimate.

Similarly, the line coherence between the H1 and H2 interferometers (Figure 3.3) has been reduced or removed for almost all lines. (The coherence between H1 or H2 and L1, not shown, is unaffected.) This can be most clearly seen by considering the accumulated coherence; lines appear as steps in the accumulation. For H1 and H2, those steps have been reduced or eliminated. Furthermore, the accumulation demonstrates that there has been no significant broadband coherence added to the interferometers. The net effect has been a reduction in the total accumulated coherence between the interferometers; in this case halving the non-accidental coherence between the interferometers, with much of the remaining coherence attributable to low frequencies.

3.4 Stochastic background S1 upper limit

The first LIGO Science run, S1, provided the data used by a number of *Upper Limits* groups to produce the first astrophysical results from the observatories—a set of upper bounds on the strength of various astrophysical sources.

One such astrophysical source is the hypothesised cosmological stochastic gravitational wave background, produced in the early universe. It is the gravitational wave analogue of the more familiar electromagnetic Cosmological Microwave Background. To detect the stochastic (gravitational wave) background requires an analysis of correlations between more than one de-

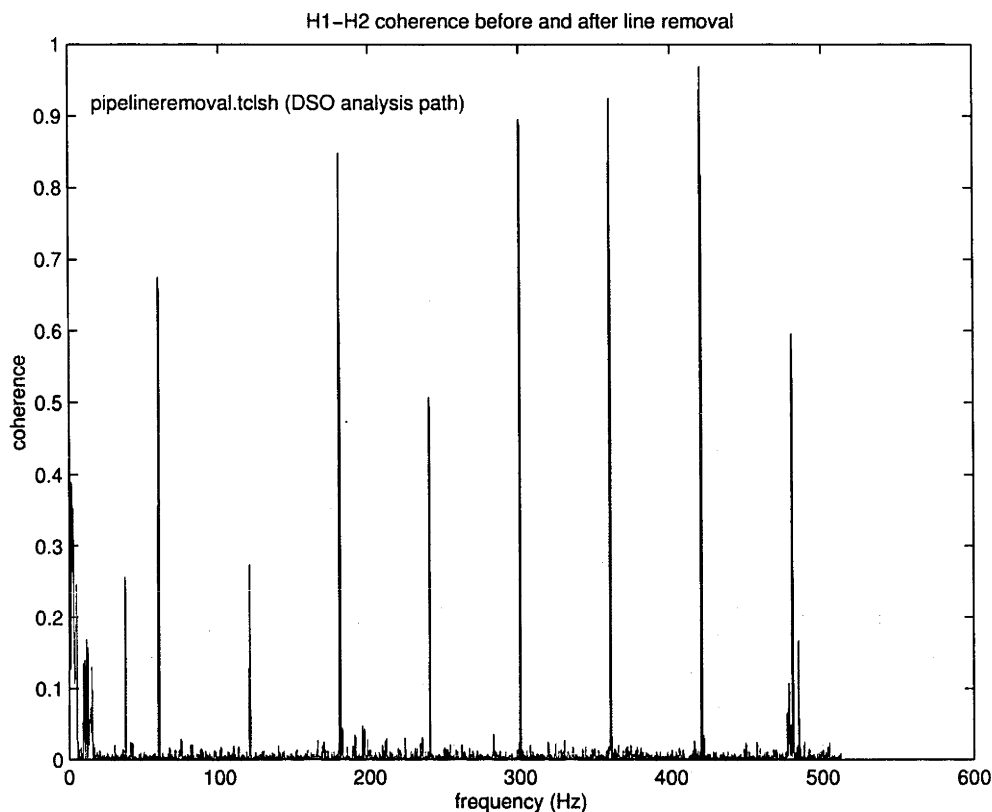


Figure 3.16: H1-H2 coherence with (red) and without (blue) the line removal stage of the stochastic pipeline.

tector, and as such the search was feared to be particularly susceptible to correlated noise sources. In particular, there were concerns for the reliability of the analysis for the co-located H1 and H2 interferometers. Spectral lines were a major source of correlations, and the stochastic background search code was an early target for line removal.

For the S1 analysis, the data analysis ‘pipeline’ included the data conditioning API, which computed various statistical properties of the data before passing its results onto the search kernel. (Full details of the S1 analysis are available in [14].) The optional line removal phase could condition the data as soon as it arrived in the data conditioning API, and before any statistics were computed.

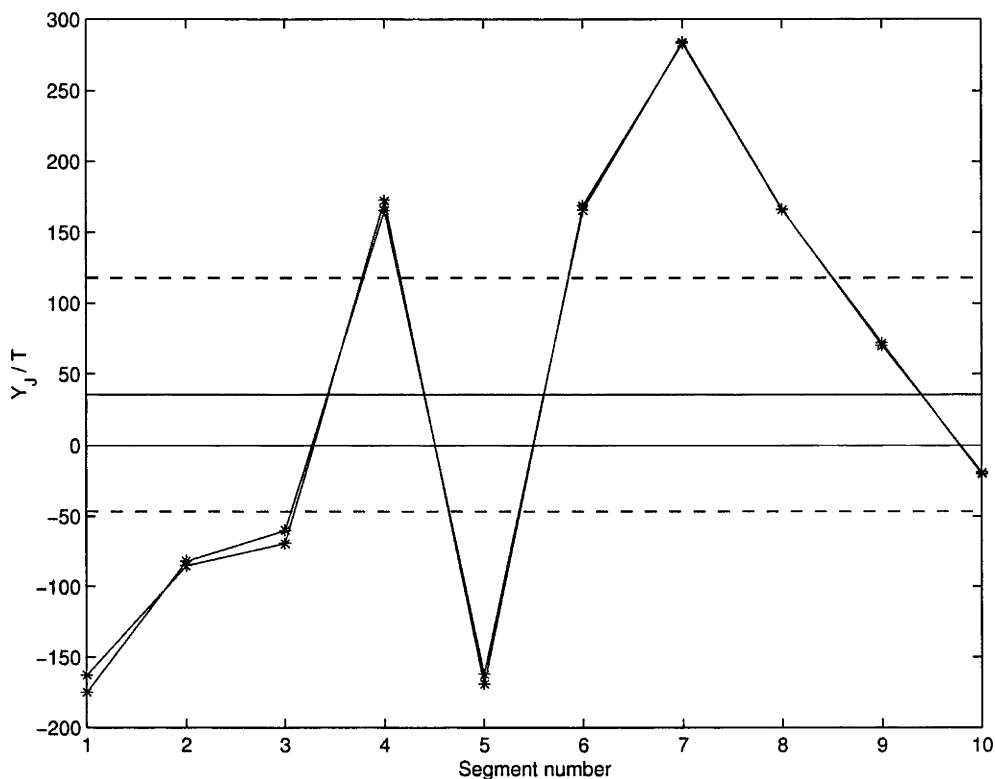


Figure 3.17: Per-data-segment (\times) and total (horizontal lines) upper limit results with and without line removal, showing no significant differences. The dashed lines are 90% confidence bounds on the (solid line) limit.

Despite the seemingly substantial reduction in coherence between H1 and H2, visible in Figure 3.16, there was no significant difference between the upper limits computed with and without line removal.

To confirm this apparent insensitivity (Figure 3.17) of the stochastic background search to correlated spectral lines, artificial spectral lines were injected into the data (Figure 3.18). Again, the upper limit was not significantly affected.

The upper limit's insensitivity to spectral lines, though advantageous, is somewhat counter-intuitive. It is not that the upper limit is insensitive to correlations, but rather that the high power of those lines results in the contribution of correlations at those frequencies being strongly suppressed.

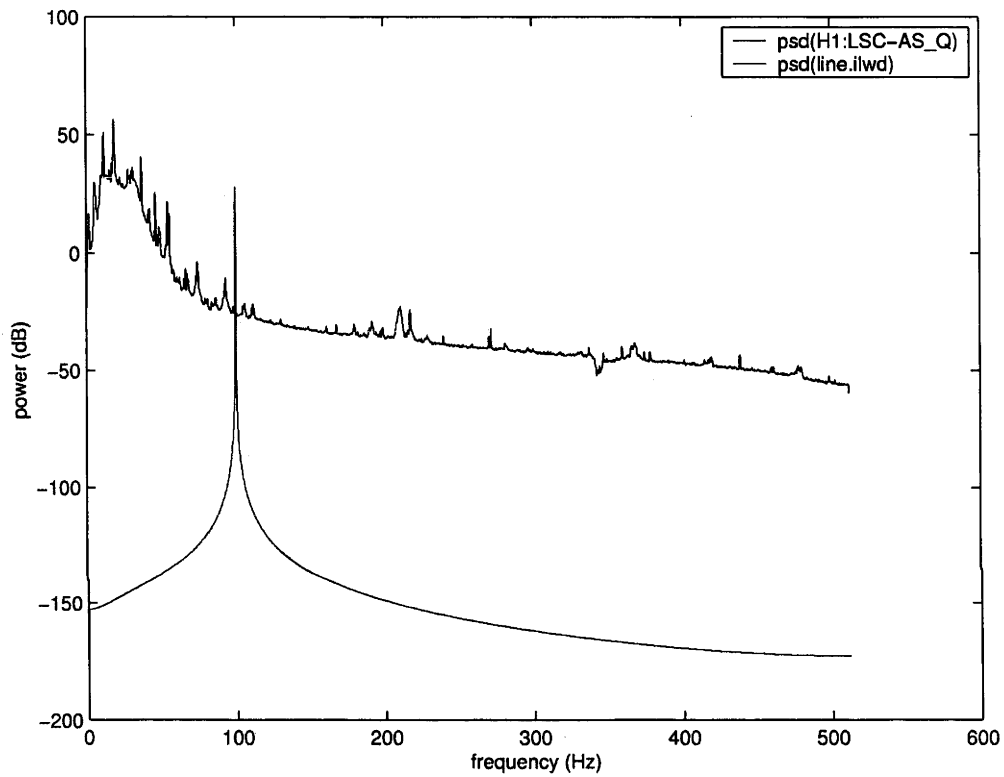


Figure 3.18: The power spectrum of the injected line (red) compared to that of the 4 km Hanford interferometer (blue).

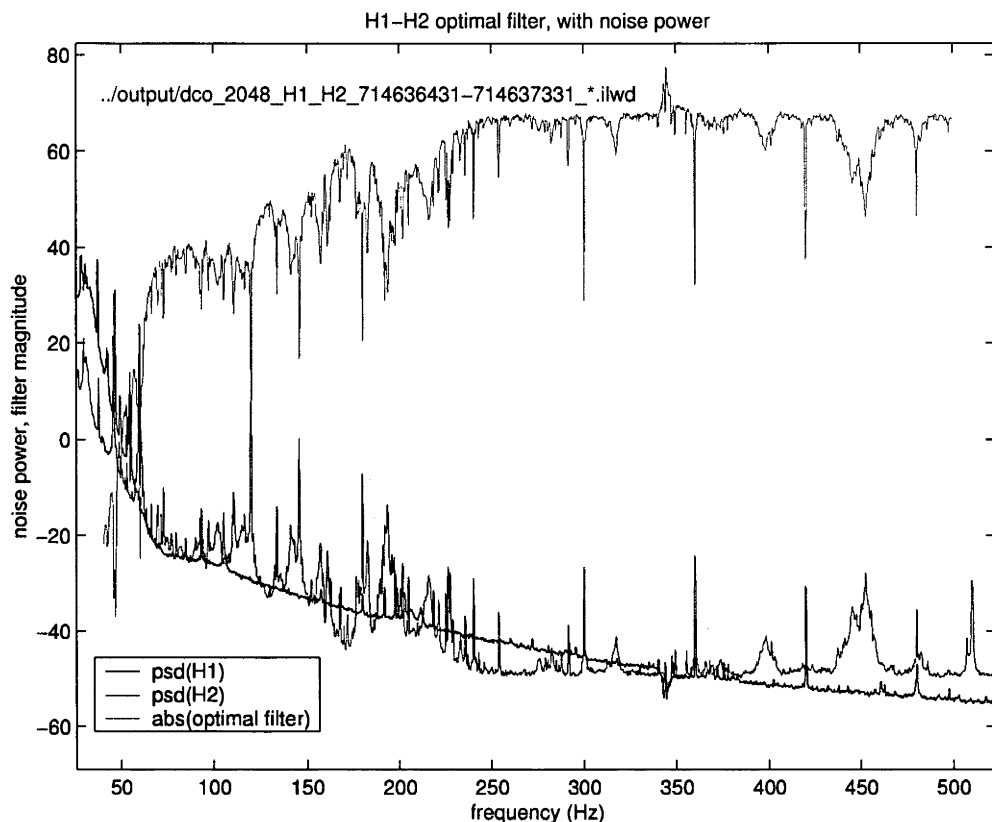


Figure 3.19: Structural comparison of the search’s optimal filter (top) and H1 and H2 spectra (bottom), showing that deep notches in the filter correspond to spectral lines.

The optimal filter employed by the search algorithm, shown in Figure 3.19, has strong dips corresponding to the spectral lines, effectively ‘notch-filtering’ out the line frequencies. Correspondingly, the line removal stage was not employed in the production of the major S1 upper limit result [14].

The line removal and injection studies, undertaken as part of the S1 analysis, demonstrated that the upper limit was insensitive to correlated spectral lines, and demonstrated that the lines would not impede any upper limits analysis of H1 and H2 correlations. (It was ultimately broad-band acoustic noise which prevented an H1-H2 upper limit.)

3.5 Conclusion

The data conditioning API `oelslr` line removal algorithm provides a non-intrusive way for LIGO astrophysical searches to reduce the power and inter-instrumental coherence of spectral lines attributed to interference from the electrical supply.

Building upon the existing functionality of the data conditioning API, the line remover provides a general framework for time-domain system identification techniques, and implements a simple class of linear model. One particular advantage of the model type chosen is that after the estimation stage is conducted, the prediction depends only on an environmental monitor, and is linearly summed with the gravitational wave channel, so that the line remover cannot interact with any incoming gravitational wave signal.

Testing of the line remover on interferometer data taken during the S1 science run have shown the method to be quite robust with respect to the selection of its free parameters. Lines are removed to the level of background noise; around the lines the background noise remains practically undisturbed. Particularly gratifying is the recovery of injected signals; other line removal methods [17] can suppress signals as well as the line. The good performance of the code, and in particular the good performance in the limit of a trivial order 1 model, indicate that potential problems such as nonlinear interference do not limit the method under realistic conditions. It had also been feared that the lines could inject small but cumulatively significant correlations in their bands of operation, but the accumulated coherence results again indicate that this is not the case.

A line removal stage was added to the stochastic background search for the S1 analysis [14]. By executing the search with and without an active line removal stage, and also with and without the injection of artificial lines, the expected but counter-intuitive robustness of the search against spectral lines was conclusively established. This increased confidence in the validity of the upper limit on the strength of an astrophysical stochastic background of gravitational waves.

Part II

Network Simulation

Chapter 4

Network simulation

Even after the launch of the four NASA orbital ‘great observatories’ [22], most electromagnetic astronomy is still affected by the geometry of the Earth. The latitude of a terrestrial optical telescope dictates how much of the northern and southern skies are visible, and the sidereal rotation of the Earth is mirrored in the ‘hours’ of right ascension of the celestial coordinate system.

For gravitational wave observatories, the geometry and rotation of the Earth have an analogous impact. In comparison to an optical or radio telescope, an interferometric gravitational wave observatory is only weakly directional, but detectors have the advantage of being able to detect sources through the Earth. This weak directionality determines the (sidereal) average sensitivity of an observatory to sources at various declinations; it also means that observatories in different locations will have different responses to the same gravitational wave.

When multiple gravitational wave observatories are combined into a single instrument, in a process analogous to aperture synthesis in radio astronomy, the relative locations of the observatories become important. The baselines between the detectors are important for triangulation of gravitational wave sources. The relative orientations of the antenna patterns of observatories affects their ability to detect any particular signal. As both the baselines and antenna patterns are dictated by the siting, these properties cannot be decoupled and independently optimised.

The detectability of a population of gravitational wave sources depends not only on these geometric factors, but also on the data analysis strategy employed. A single detector may be considered in an analysis, or data from multiple detectors may be analysed cooperatively. If data from multiple

detectors is to be considered, the data may be combined coherently or incoherently. An analysis may be limited by available computational power or bandwidth.

A great deal of effort has gone into optimising the performance of these analyses, but scant work has been done on optimising another component in the sensitivity: the siting of the component detectors. Next-generation gravitational wave observatories, such as the proposed Australian Interferometric Gravitational Observatory (AIGO), should be located so as to optimise both their capacity for individual discovery and their contribution to the global network of gravitational wave detectors.

This chapter presents a strategy for computing *figures of merit* with which to compare systems of gravitational wave detectors and analyses, and the common constructions for the specific figures of merit and analyses introduced in Chapters 5 and 6.

4.1 Geometrical considerations

4.1.1 Interferometric gravitational wave detectors

Consider a reference frame co-rotating with the Earth. Define twin cartesian $[x \ y \ z]$ and spherical polar $[r \ \theta \ \phi]$ coordinate systems with their origins at the centre of the Earth (assumed to be a perfect sphere). Then

$$[x \ y \ z] = [r \cos \theta \cos \phi \ r \cos \theta \sin \phi \ r \sin \theta], \quad (4.1)$$

where θ and ϕ correspond to latitude North and longitude East respectively (in radians). Along any line of constant θ and ϕ , the orthonormal unit vectors *local North* $\hat{\theta}$ and *local East* $\hat{\phi}$ may be defined in cartesian coordinates:

$$\begin{aligned} \hat{\theta} &= [-\sin \theta \cos \phi \ -\sin \theta \sin \phi \ \cos \theta], \\ \hat{\phi} &= [\quad -\sin \phi \quad \cos \phi \quad 0]. \end{aligned} \quad (4.2)$$

A horizontal interferometric gravitational wave observatory at sea level with mutually perpendicular arms of equal length may be described by its latitude θ , longitude ϕ and the *orientation* angle ψ of its arms clockwise from local North. The unit vectors along the arms are

$$\begin{aligned} \hat{e}_x &= \hat{\theta} \cos \psi + \hat{\phi} \sin \psi, \\ \hat{e}_y &= \hat{\theta} \sin \psi - \hat{\phi} \cos \psi, \end{aligned} \quad (4.3)$$

and the ideal linear response m of such a detector [23] to incident strain \mathbf{H} is given by

$$m = \sum_{i,j=1}^3 R_{ij} H_{ij} \quad (4.4)$$

where

$$\mathbf{R} = \hat{\mathbf{e}}_x^T \hat{\mathbf{e}}_x - \hat{\mathbf{e}}_y^T \hat{\mathbf{e}}_y. \quad (4.5)$$

4.1.2 Gravitational wave sources

Similarly to an interferometer, a source of gravitational radiation can be instantaneously described [23] in terms of the latitude θ and longitude ϕ for which it is overhead (i.e., it lies on the line of sight θ, ϕ) and an orientation angle ψ (required to uniquely determine the polarisations) from North $\hat{\theta}$. The x and y axes of the source are then as in Equation 4.3, producing a *polarisation basis*

$$\begin{aligned} \hat{\mathbf{E}}_+ &= \hat{\mathbf{e}}_x^T \hat{\mathbf{e}}_x - \hat{\mathbf{e}}_y^T \hat{\mathbf{e}}_y, \\ \hat{\mathbf{E}}_\times &= \hat{\mathbf{e}}_x^T \hat{\mathbf{e}}_y + \hat{\mathbf{e}}_y^T \hat{\mathbf{e}}_x \end{aligned} \quad (4.6)$$

and the time-dependent strain $\mathbf{H}(t)$ produced by plane gravitational waves may be described in this basis by the two functions of time h_+ and h_\times , so that

$$\mathbf{H}(t) = h_+(t) \hat{\mathbf{E}}_+ + h_\times(t) \hat{\mathbf{E}}_\times \quad (4.7)$$

in the limit where $t \ll T_{\text{sidereal}}$, i.e., when the rotation of the Earth does not significantly change the relative orientations of the source and detector over the duration of the signal.

When this is not the case, (i.e., for continuous gravitational wave sources), the variation in real-time orientations must be taken into consideration. One way to achieve this is to treat the response as a function of time, $\mathbf{R}(t)$. This is the approach taken in Chapter 6.

4.1.3 Antenna patterns

The *antenna patterns* [23]

$$F_+ = \sum_{i,j=1}^3 R_{ij} (\hat{\mathbf{E}}_+)_{ij}, \quad (4.8)$$

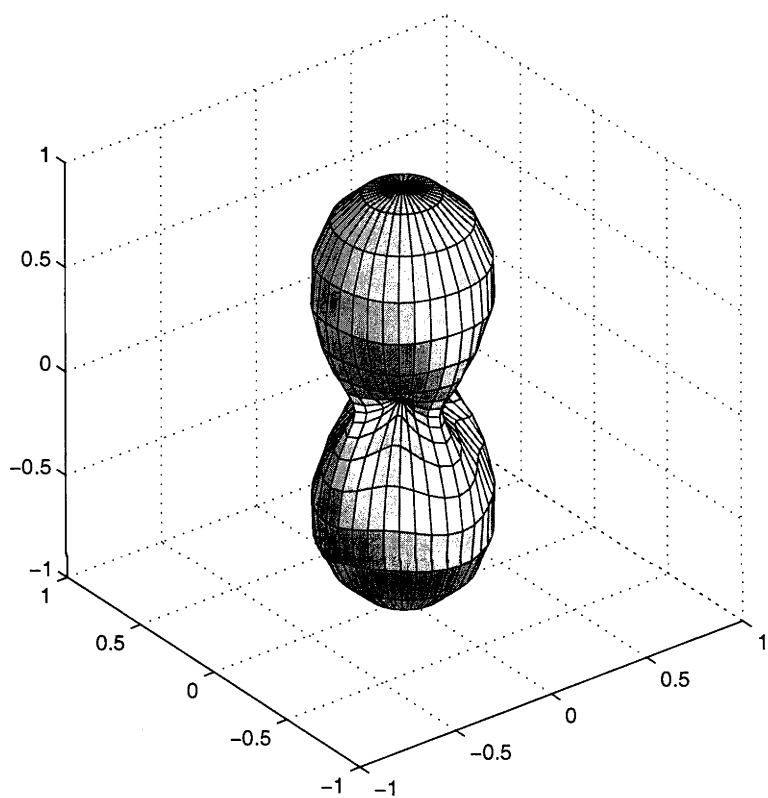


Figure 4.1: Antenna pattern $(F_+^2 + F_\times^2)(\theta, \phi)$ of an ideal interferometric gravitational wave detector with arms $\hat{e}_x = \hat{x}$ and $\hat{e}_y = \hat{y}$.

$$F_{\times} = \sum_{i,j=1}^3 R_{ij}(\hat{\mathbf{E}}_{\times})_{ij}, \quad (4.9)$$

are a particular detector's response to the '+' or '×' polarisations. The detector's response to a signal described in that basis by $h_{+}(t)$ and $h_{\times}(t)$ is thus given by

$$m(t) = F_{+}h_{+}(t) + F_{\times}h_{\times}(t). \quad (4.10)$$

The quantity

$$F_{+}^2 + F_{\times}^2 \quad (4.11)$$

is independent of the choice of polarisation basis; it corresponds to the relative power received, from an unpolarised source in a particular direction, by an ideal detector. It is plotted in Figure 4.1.

Peak sensitivity occurs when the source is perpendicular to the plane of the arms; for a terrestrial detector, this corresponds to a source directly above or below. The detector is insensitive along the 'arm diagonal' directions $\pm\hat{\mathbf{e}}_x \pm \hat{\mathbf{e}}_y$, where symmetry dictates that the strain on each arm is equal.

4.1.4 Implementation

The formulation of response in terms of matrix operations facilitates its implementation in the matrix-based MATLAB language.

Listing 4.1: Compute the response matrix of a gravitational wave detector.

```
function out = detector(theta, phi, psi)
%DETECTOR Response matrix for an ideal interferometric gravitational
  wave detector
%
% DETECTOR(THETA, PHI, PSI) returns a detector object with location
  members X, Y, Z in Cartesian and THETA, PHI in spherical polar
  coordinates, arm orientation PSI and a RESPONSE matrix to incident
  gravitational waves.
%
% See also SOURCE, RESPONSE.

up = [ cos(theta) * cos(phi), cos(theta) * sin(phi), sin(theta)];
north = [- sin(theta) * cos(phi), - sin(theta) * sin(phi), cos(theta)];
east = [- sin(phi), cos(phi), 0];
```

```

out.x = east * cos(psi) + north * sin(psi);
out.y = - east * sin(psi) + north * cos(psi);
out.z = up;

out.response = 0.5 * (out.x'*out.x - out.y'*out.y);

out.theta = theta;
out.phi = phi;
out.psi = psi;

```

Similarly, the polarisation basis for a gravitational wave source may be constructed.

Listing 4.2: Compute the polarisation basis of a source.

```

function [plus, cross] = source(theta, phi, psi)
%SOURCE Polarisation basis strains for a gravitational wave source
%
% SOURCE(THETA, PHI, PSI) returns polarisation basis strains [PLUS,
%   CROSS] for a source on the ray (THETA, PHI) with orientation PSI
%
% See also DETECTOR, RESPONSE.

north = [- sin(theta) * cos(phi), - sin(theta) * sin(phi), cos(theta)];

east = [- sin(phi), cos(phi), 0];

x = east * cos(psi) + north * sin(psi);
y = - east * sin(psi) + north * cos(psi);

plus = (x'*x - y'*y);
cross = (x'*y + y'*x);

```

To determine the amplitude of an ideal detector response to an incident polarisation we sum over the elements of the response matrix.

Listing 4.3: Compute the response of a detector to a given strain.

```

function out = response(d, strain)
%RESPONSE The response of an ideal detector to incident strain

```

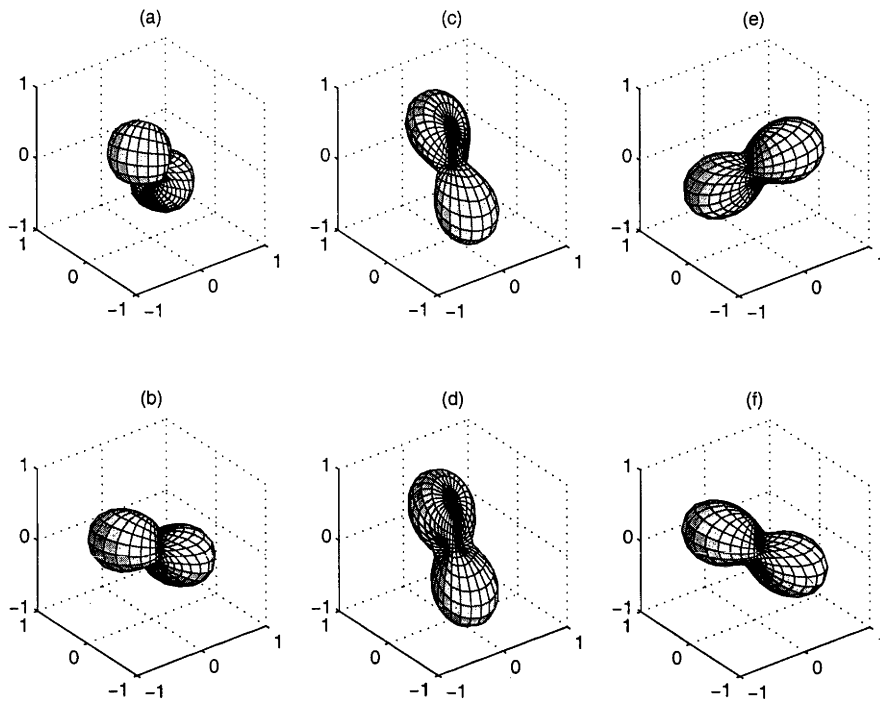


Figure 4.2: Antenna patterns of existing detectors (a) LIGO Hanford (both instruments), (b) LIGO Livingston, (c) VIRGO, (d) GEO, (e) TAMA and proposed detector (f) AIGO.

```

%
% RESPONSE(D, STRAIN) returns the response of detector D to strain S.
%
% See also DETECTOR, SOURCE.

out = sum(sum(d.response .* strain)); % Note that ".*" performs
    componentwise (not matrix) multiplication

```

4.1.5 Existing and proposed detectors

The geometrical properties of existing interferometric gravitational wave detectors have been collated by Allen [24]¹. Scripts were created to facilitate the inclusion of detectors with the properties of real observatories in the simulations.

Listing 4.4: Compute the response of a real observatory.

```
function out = lho
%LHO returns a detector object corresponding to LIGO Hanford
  Observatory
%
% Latitude 46.45 north, longitude 119.41 west, and arm orientation 36.8
  degrees counter-clockwise from north
%
% See also DETECTOR, LLO, VIRGO, GEO, TAMA, AIGO.

out = detector(radians(46.45), radians(-119.41), radians(36.8));
```

The antenna patterns of the different detectors may be compared in Figure 4.2.

4.2 Figures of merit

We will call a system of interferometric gravitational wave observatories and their co-operative data analysis technique a *network*, and present a simple formalism providing a general basis for the comparison of networks under certain criteria. Computationally-amenable *figures of merit* approximating significant properties of the network—such as the rate of detections produced by a network for a given source population under a given co-operative data analysis technique—are used to rank the relative performance of different networks.

Consider the set \mathcal{N} of all networks. A *figure of merit* f is defined as a real function on some subset \mathcal{S} of networks, $f : \mathcal{S} \subseteq \mathcal{N} \rightarrow \mathbb{R}$, for which $f(\alpha) > f(\beta)$ (where $\alpha \in \mathcal{S}$ and $\beta \in \mathcal{S}$) is interpreted as the statement that network α is better than network β . An example of a figure of merit is the

¹To maintain compatibility with [24] the scripts 4.1–4 use slightly different conventions to those presented in subsections 4.1.1–3.

inverse of the cost of the construction of a network. There are many possible figures of merit and, in general, they will not produce the same rankings; assessment of the significance of the results of different figures of merit must be done by assessing the significance of the figures of merit themselves.

It is important, also, that figures of merit be computationally tractable as well as significant. The restriction of the domain of the figure of merit to a particular subset $\mathcal{S} \subseteq \mathcal{N}$ of networks may simplify computation of the figure of merit while still permitting the examination of problems of interest. Frequently, this involves finding the optimal network or networks $\hat{\alpha}$ in a set $\mathcal{T} \subseteq \mathcal{S}$ which is determined by the constraints on the problem,

$$\hat{\alpha} = \{\beta : \beta \in \mathcal{T}, f(\beta) = \max_{\mathcal{T}} f\}. \quad (4.12)$$

Here we restrict ourselves to considering subsets $\mathcal{S} \subseteq \mathcal{N}$ of networks of fixed numbers of interferometric gravitational wave detectors, where all detectors in \mathcal{S} are assumed to be identical. A subset \mathcal{S} is completely described by the number n of detectors in a network, the ‘design’ Θ of the identical detectors and the co-operative analysis method Ξ used. Any particular network $\alpha \in \mathcal{S}$ is then completely described by, for each detector, the latitudes θ_i and longitudes ϕ_i of the beam-splitters, and the orientation angles ψ_i of the x-arms counter-clockwise from North (under the assumption of horizontal detectors on a perfectly spherical Earth). Each network is then a point in the $3n$ -dimensional parameter space $[(\theta_1, \phi_1, \psi_1), \dots, (\theta_n, \phi_n, \psi_n)]$.

We will consider *families* of figures of merit $f_{(n,\Theta,\Xi)}$; these permit comparisons of different geographical configurations of networks in a subset \mathcal{S} , but *not* comparisons of networks with different numbers of detectors n , designs Θ or analysis algorithms Ξ .

4.3 Summary

An interferometric gravitational wave detector’s sensitivity to incoming gravitational radiation is dependent upon the relative orientation of detector and source. When a detector is restricted to lie on the Earth’s surface, this antenna pattern is dictated by the siting of the instrument. The response of an ideal detector to a strain polarisation can be modelled with some simple vector algebra.

To answer questions about the relative merits of different sites or parameters of detectors or networks of detectors, a simple figure of merit formalism

can be used as a basis. In Chapter 5 we consider the merit of different configurations of observatories around the Earth for the purpose of detecting binary inspiral events by either coincident or coherent data analysis strategies. In Chapter 6, we examine how the latitude of detectors impacts upon their ability to detect a galactic population of sources of continuous gravitational waves.

Chapter 5

Geographical configuration

The theoretically-known waveforms of the inspiral phase of merging binary compact stellar systems is one of the most promising sources for first-generation terrestrial interferometric observatories. The events are rare, brief and predominantly faint. Distinguishing between real signals and instrumental artifacts is the limiting factor. Combining data from multiple observatories can improve both sensitivity and confidence, by weeding out such artifacts. In fact, the dual-detector LIGO design embodies the opinion that detection of an event by at least two independent instruments is required for the widespread acceptance of a claim. Network analysis is the generalisation of this concept.

The coincident network analysis technique [25, 26], in its simplest form, allows independent searches to be performed by each detector in the network; a signal is only detected by the network when the signal is detected by each member detector. A more recently proposed technique is *coherent* network analysis [27, 28], whereby the output of all detectors is collected and then a single search is performed on the combined data. The coherent network analysis has a theoretical advantage over the coincident network analysis, but the practical merits of each are still under debate.

The twin LIGO sites were chosen to facilitate a coincidence analysis—they are distant enough to reduce common environmental disturbances and produce a measurable arrival time difference, but close enough to have similar antenna patterns [23] and so produce similar responses to an incident gravitational wave. Likewise, the location of the proposed Australian-International Gravitational Observatory (AIGO) [29] has been selected to be near-antipodal to the LIGO sites and thus share their antenna patterns, whilst introducing

a significant arrival time delay [30]. The site of VIRGO, however, and the proposed Laser Cryogenic Gravitational Telescope (LCGT) were not selected [31, 32] to facilitate a global network analysis. This implies that any realistic global network will likely be, in this sense, sub-optimal and we will determine how significantly this will impact the ability of the network to do science.

Questions of how to optimally configure the global network arise in this context. We describe a formalism for comparing different geographical configurations of a global network of interferometric gravitational wave observatories, using both the coincident network analysis method and the coherent network analysis method. We have constructed a network model to compute a figure of merit based on the relative detection rate for the particular case of a uniform population of standard-candle binary inspirals.

The increasing viability of the new coherent network analysis technique [27, 28] encourages us to reconsider existing results about the global network; in particular, the influence of instrument siting on the quality of the network as a whole [26].

5.1 Detection of binary inspiral events

We define a particular figure of merit corresponding to the detection rate for a population of standard-candle binary inspiral events.

Consider a particular class of binary inspiral systems, producing a particular deterministic gravitational waveform. Distribute these systems uniformly in flat space and randomly orient them. Let the distribution be unchanging in time so that any volume of space produces a constant rate of events. The property on which we will base the figure of merit $f_{(n,\Theta,\Xi)}$ is the rate at which events from this population may be confidently detected by the application of some network analysis algorithm Ξ to any given network of n gravitational wave detectors of design Θ .

5.1.1 Waveform and response

A simple binary inspiral [23] produces a quadrupole strain of the form

$$\mathbf{H} = \frac{\eta(t)}{r} (\hat{\mathbf{E}}_+ h_+(t) + \hat{\mathbf{E}}_\times h_\times(t)), \quad (5.1)$$

where

$$h_+(t) = (1 + \cos^2 i) \cos \zeta(t), \quad (5.2)$$

$$h_\times(t) = 2 \cos i \sin \zeta(t), \quad (5.3)$$

and r is the distance traversed by the gravitational radiation, i is the inclination angle of the source to the line of sight, and η and ζ depend on other properties of the emitting system (and are unaffected by the system's distance and orientation with respect to the component detectors). Note that $\eta(t)$ is the envelope of the more-rapid $\sin \zeta(t)$ and $\cos \zeta(t)$ oscillations; the structure of $\eta(t)$ and $\zeta(t)$ beyond this is not relevant to the rest of our analysis [23].

The response of any single detector in the network to this strain is

$$m(t) = \frac{\eta(t)}{r} [(1 + \cos^2 i) \cos \zeta(t) F_+ + 2 \cos i \sin \zeta(t) F_\times], \quad (5.4)$$

where the antenna patterns and source inclination F_+ , F_\times , and i encode the relative orientations of the emitting system and detector.

5.1.2 Analysis strategies

The output $g(t)$ of the detector also consists of noise $n(t)$, assumed to be additive with the signal response and stationary on the timescale of the signal,

$$g(t) = m(t) + n(t). \quad (5.5)$$

The noise component of the detector output can be made Gaussian by the application of a linear whitening filter ($'$), also stationary on the timescale of the signal,

$$g'(t) = m'(t) + n'(t). \quad (5.6)$$

The application of the filter also alters the response.

Following Finn [27], to determine with confidence if a particular signal response $m(t)$ is present in the filtered output $g'(t)$ of a single detector, consider the mutually exclusive hypotheses H_0 , that the detector output consists solely of Gaussian noise $g'(t) = n'(t)$, and H_m , that the detector output consists of the sum of Gaussian noise and the filtered signal response,

$g'(t) = n'(t) + m'(t)$. The *likelihood ratio* Λ is then the ratio of the probabilities of the observed output $g'(t)$ arising under each hypothesis,

$$\Lambda(g', m') = \frac{P(g'|H_m)}{P(g'|H_0)} \quad (5.7)$$

$$= \frac{P(g' - m'|H_0)}{P(g'|H_0)}. \quad (5.8)$$

The likelihood may be readily computed by *matched filtering* [27],

$$\ln \Lambda(g'|m') = 2\langle g', m' \rangle - \langle m', m' \rangle, \quad (5.9)$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product of the two time series. This allows us to determine the ‘plausibility’ that the detector output arose from any particular signal response.

The *maximum* likelihood

$$\Lambda_{\max}(g'|\mathcal{M}) = \max_{m \in \mathcal{M}} \Lambda(g'|m') \quad (5.10)$$

is the likelihood of the most plausible signal response \hat{m} in some set of responses \mathcal{M} . A *confident detection* of a candidate signal $\hat{m} \in \mathcal{M}$ is said to have occurred when

$$\Lambda_{\max}(g'|\mathcal{M}) = \Lambda(g'|\hat{m}') > \Lambda_0 \quad (5.11)$$

where Λ_0 is a *threshold* value that is set sufficiently high to ensure that when no signal is present it is exceeded only at an acceptable *false alarm* rate. A *false dismissal* occurs when the likelihood for a weak but real signal fails to exceed the threshold.

A simple *coincident* network analysis can be performed using only the above algorithm on each detector: a detection occurs only when each observatory detects a signal \hat{m}_i . This requirement allows the thresholds Λ_i to be lower than for a single detector, as more frequent false alarms are ‘vetted’ by other detectors. For a network of identical detectors, the thresholds themselves are identical, so that a detection occurs when

$$\min \Lambda_{\max}(g'_i|\mathcal{M}) > \Lambda_{\text{coincident}}. \quad (5.12)$$

Alternatively, *coherent* network analysis vectorises the maximum likelihood test to treat the network as a whole, a confident detection occurring when

$$\Lambda_{\max}(\mathbf{g}'|\mathcal{M}) > \Lambda_{\text{coherent}}, \quad (5.13)$$

where $\mathbf{g}' = [g_0 \dots g_n]$. No single detector is required to meet any threshold. This technique is theoretically optimal in the same sense as the maximum likelihood test is optimal for a single detector. When the noise is uncorrelated between detectors in the network, the likelihood is separable, so that

$$\Lambda(\mathbf{g}'|\mathbf{m}) = \prod_{i=1}^n \Lambda(g'_i|m_i), \quad (5.14)$$

but as the maximisation occurs for the system as a whole, the individual signal responses m_i typically do not correspond to maximum likelihoods for the individual detectors [27],

$$\max_{m_i \in \mathcal{M}} \prod_{i=1}^n \Lambda(g'_i|m_i) > \Lambda_{\text{coherent}}. \quad (5.15)$$

5.1.3 Detection rate

We are concerned only with the case where a physical signal is present, as false alarms have been limited to an acceptably low rate.

Consider the gravitational wave signal from a particular binary inspiral event, with all parameters fixed except its distance to the detectors (corresponding to the inverse amplitude of the wave: Equation 5.4). We may establish an effective maximum distance r_{max} beyond which the probability of detecting such a source falls below some threshold. This value could be computed from the definitions of the tests above, for example, by Monte Carlo simulation.

Consider a population of otherwise identical binary inspiral systems uniformly distributed in (flat) space and randomly oriented. The effective volume V of space in which the events can be detected can be computed from r_{max}^3 by integrating over the sky and averaging over source orientation and inclination,

$$V = \frac{1}{4\pi} \int_{\Omega} r_{\text{max}}^3 \cos \theta \sin i d\Omega. \quad (5.16)$$

For a constant event rate per unit volume ρ , the rate of confident detections from the network is ρV .

This constitutes a valid figure of merit $f = \rho V$. A network with a higher rate of detections (for the same level of confidence) is clearly better than a

network with a lower rate of detections, at least so far as detection of this particular class of binary inspirals is concerned.

This figure of merit is, however, prohibitively expensive to compute naively. Instead, we simplify it and introduce approximations to implement a new, computable, figure of merit.

5.1.4 Implementation

From Equation 5.4,

$$\begin{aligned} m'(t) &= [\eta(t) \cos \zeta(t)]' \frac{1 + \cos^2 i}{r} F_+ \\ &+ [\eta(t) \sin \zeta(t)]' \frac{2 \cos i}{r} F_\times. \end{aligned} \quad (5.17)$$

Noting that

$$\langle (\eta \cos \zeta)', (\eta \cos \zeta)' \rangle \approx \langle (\eta \sin \zeta)', (\eta \sin \zeta)' \rangle, \quad (5.18)$$

$$\langle (\eta \cos \zeta)', (\eta \sin \zeta)' \rangle \approx 0, \quad (5.19)$$

then when a signal m' is present

$$\begin{aligned} \ln \Lambda(g'|m') &= 2\langle m' + n', m' \rangle - \langle m', m' \rangle \\ &= \langle m', m' \rangle + 2\langle n', m' \rangle \end{aligned} \quad (5.20)$$

$$\begin{aligned} &\approx \langle (\eta \cos \zeta)', (\eta \cos \zeta)' \rangle \\ &\times \frac{1}{r^2} [(1 + \cos^2 i)^2 F_+^2 + 4 \cos^2 i F_\times^2] \\ &+ 2\langle n', m' \rangle. \end{aligned} \quad (5.21)$$

We assume that for confident detections

$$\ln \Lambda_{\max}(g') \approx \ln \Lambda(g'|m') \approx \overline{\ln \Lambda(g'|m')}, \quad (5.22)$$

in other words, that the most plausible signal approximates the real signal, and that the contribution of noise to the likelihood is negligible.

Under this assumption, the coincident test in Equation 5.12 becomes

$$\begin{aligned} \ln \Lambda_{\text{coincident}} &< \min_i \ln \overline{\Lambda(g'_i|m_i)} \\ &= \frac{\langle (\eta \cos \zeta)', (\eta \cos \zeta)' \rangle}{r^2} \\ &\times \min_i [(1 + \cos^2 i)^2 (F_+)_i^2 \\ &+ 4 \cos^2 i (F_\times)_i^2], \end{aligned} \quad (5.23)$$

and the coherent test in Equation 5.15 becomes

$$\begin{aligned}
 \ln \Lambda_{\text{coherent}} &< \ln \prod_i \overline{\Lambda(g'_i | m_i)} \\
 &= \frac{\langle (\eta \cos \zeta)', (\eta \cos \zeta)' \rangle}{r^2} \\
 &\times \sum_i [(1 + \cos^2 i)^2 (F_+)_i^2 \\
 &+ 4 \cos^2 i (F_\times)_i^2]. \tag{5.24}
 \end{aligned}$$

The two tests differ only in their use of min or \sum to combine the likelihoods.

The maximum detectable distance r_{max} is the distance at which the threshold is reached; for a coincident analysis

$$\begin{aligned}
 r_{\text{max}}^2 &= \frac{\langle (\eta \cos \zeta)', (\eta \cos \zeta)' \rangle}{\ln \Lambda_{\text{coincident}}} \\
 &\times \min_i [(1 + \cos^2 i)^2 (F_+)_i^2 \\
 &+ 4 \cos^2 i (F_\times)_i^2], \tag{5.25}
 \end{aligned}$$

and for a coherent analysis

$$\begin{aligned}
 r_{\text{max}}^2 &= \frac{\langle (\eta \cos \zeta)', (\eta \cos \zeta)' \rangle}{\ln \Lambda_{\text{coherent}}} \\
 &\times \sum_i [(1 + \cos^2 i)^2 (F_+)_i^2 \\
 &+ 4 \cos^2 i (F_\times)_i^2]. \tag{5.26}
 \end{aligned}$$

Then,

$$\begin{aligned}
 V_{\text{coincident}} &\propto f_{n,\theta,\text{coincident}} \\
 &\propto \int_{\Omega} \{ \min_i [(1 + \cos^2 i)^2 (F_+)_i^2 \\
 &+ 4 \cos^2 i (F_\times)_i^2] \}^{\frac{3}{2}} \cos \theta \sin i d\Omega, \tag{5.27}
 \end{aligned}$$

$$\begin{aligned}
 V_{\text{coherent}} &\propto f_{n,\theta,\text{coherent}} \\
 &\propto \int_{\Omega} \{ \sum_i [(1 + \cos^2 i)^2 (F_+)_i^2 \\
 &+ 4 \cos^2 i (F_\times)_i^2] \}^{\frac{3}{2}} \cos \theta \sin i d\Omega, \tag{5.28}
 \end{aligned}$$

where the neglected term $\langle (\eta \cos \zeta)', (\eta \cos \zeta)' \rangle$ depends only on the source class, and the thresholds $\Lambda_{\text{coincident}}$ and $\Lambda_{\text{coherent}}$ are assumed to depend only on the detector design. We neglect the dependance of the thresholds on the geographical configuration of the network¹.

The figures of merit $f_{n,\Theta,\text{coincident}}$ and $f_{n,\Theta,\text{coherent}}$ are (granted approximations) linearly proportional to the actual rate ρV . To evaluate these figures of merit, the response matrices \mathbf{R}_i are first computed for each detector using Equation 4.5. Numerical *Monte Carlo* integration is implemented, randomly selecting source parameters from the population and evaluating the interior of the integral many times (using Equations 4.8 and 4.9 to compute the antenna patterns), and averaging the result.

The $3n$ -dimensional parameter space is too large to be computationally amenable, and so we consider only proper subsets of particular interest. For example, the best site (under our figure of merit) to augment an existing network of n detectors can be found by fixing the first n detectors of a $n + 1$ detector network, and varying only $(\theta_{n+1}, \phi_{n+1}, \psi_{n+1})$. Furthermore, noting that the figure of merit depends only weakly on the orientation², ψ_{n+1} , we can fix it to an arbitrary value, and vary only the latitude and longitude. The figure of merit over this two-dimensional section of parameter space then corresponds to a map of the relative merit of different sites on the Earth for augmenting an existing network.

5.2 Results

We may use our figures of merit, Equations 5.27 and 5.28, to answer a variety of questions about the network; we choose to determine the optimal detector to augment an existing network of identical detectors.

Formally, consider a network of n detectors. Detectors 1 to $n - 1$ represent the existing detectors with fixed latitude θ_i , longitude ϕ_i and orientation ψ_i . Detector n represents the augmenting detector with variable latitude θ_n , longitude ϕ_n and orientation ψ_n . Effectively we wish to compute the merit f over the subset $\mathcal{T} \subset \mathcal{S}_{n,\Theta,\Xi} \subset \mathcal{N}$, where \mathcal{T} represents the 3-dimensional surface of constant θ_i, ϕ_i, ψ_i for $i < n$.

¹Thanks to Peter Shawhan for pointing this out.

²See Figures 5.1, 5.2, 5.3, 5.4: at the upper and lower edges, corresponding to the north and south geographic poles, the detectors rotate in place as the longitude varies; despite this the figure of merit remains constant to a good approximation.

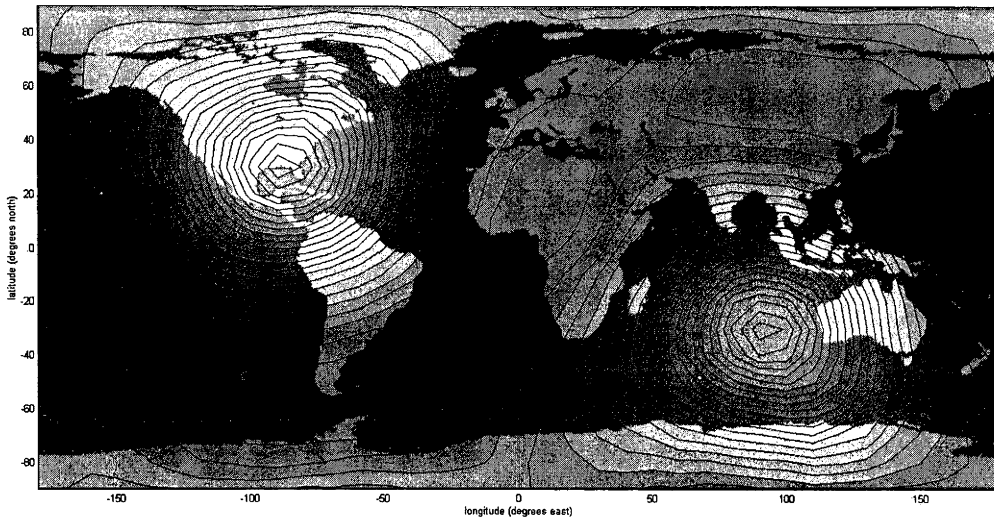


Figure 5.1: Relative merit of an additional site to augment the LIGO Livingston Observatory in a coincident analysis (lighter is better, contours every 2.5%). The minimum detection rate is 41% of the maximum.

We can further reduce \mathcal{T} by noting that $f_{\text{coincident}}$ and f_{coherent} vary only weakly with ψ_n . We may then additionally fix the orientation ψ_n at an arbitrary value, and consider only the 2-dimensional slice produced by varying θ_n and ϕ_n .

This 2-dimensional set has a straightforward interpretation as the geographical map of the merit of any site on the surface of the Earth to augment an existing network of $n - 1$ identical detectors with another such detector.

Consider first a single interferometer, at the site [24] of the LIGO Livingston Observatory (LLO). For a coincident network analysis, the merit of an additional site to augment LLO is given in Figure 5.1. It demonstrates, as expected, that sites near or near-antipodal to LLO are best to augment it. This is the rationale behind the siting of the LIGO detectors, and the proposed AIGO detector. The worst configurations produce a substantially reduced detection rate; approximately 40% that of the optimal configuration. Unfortunately, the locations of VIRGO and the proposed LCGT fall into this category.

It is interesting to note that for this simple case the map bears some resemblance to the “peanut” antenna pattern of the fixed single detector;

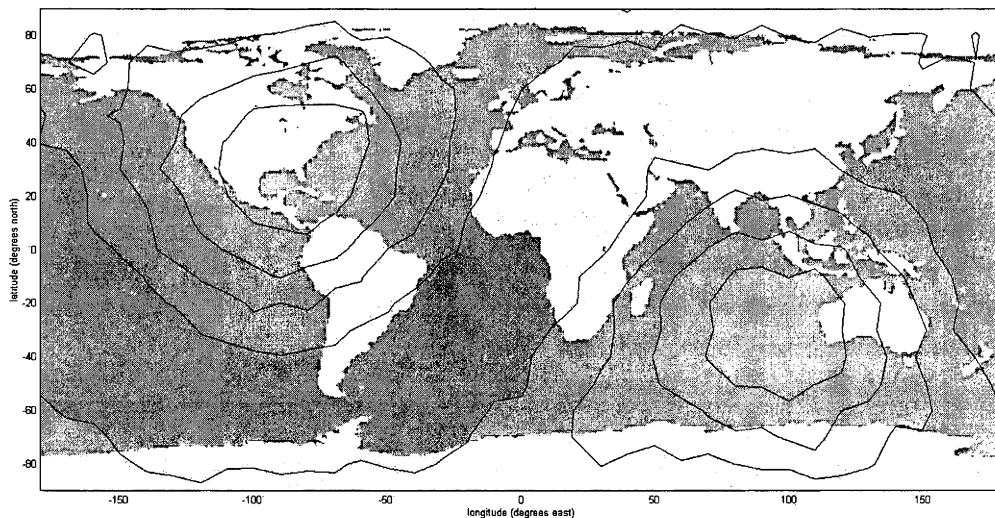


Figure 5.2: Relative merit of an additional site to augment the LIGO Livingston Observatory in a coherent analysis (lighter is better, contours every 2.5%). The minimum detection rate is 89% of the maximum.

the weak directionality of the varying detector, and the superiority of a co-aligned network [33] are responsible for this effect. This resemblance breaks down for more complicated networks.

Considering the same configuration of a fixed LLO detector and a varying detector with a coherent network analysis in Figure 5.2, the qualitative structure of the map is similar, but quantitatively it is quite different. For a coherent analysis, the worst configurations produce a detection rate that is still 90% of optimal; site merit does not vary substantially with location.

We now move on to consider an approximation to the existing global network of the larger interferometric gravitational wave detectors. We model the LIGO-VIRGO network as three identical interferometers at the sites of LIGO Hanford Observatory (LHO), LIGO Livingston Observatory and VIRGO [24]. Note that this model neglects the 2 kilometre LHO instrument, and the differences between the LIGO and VIRGO instruments. Similarly, we augment this three-detector network with a fourth (identical) detector at different locations and compare the relative detection rates of the resulting network.

Using a coincident network analysis in Figure 5.3, we see that the merit of

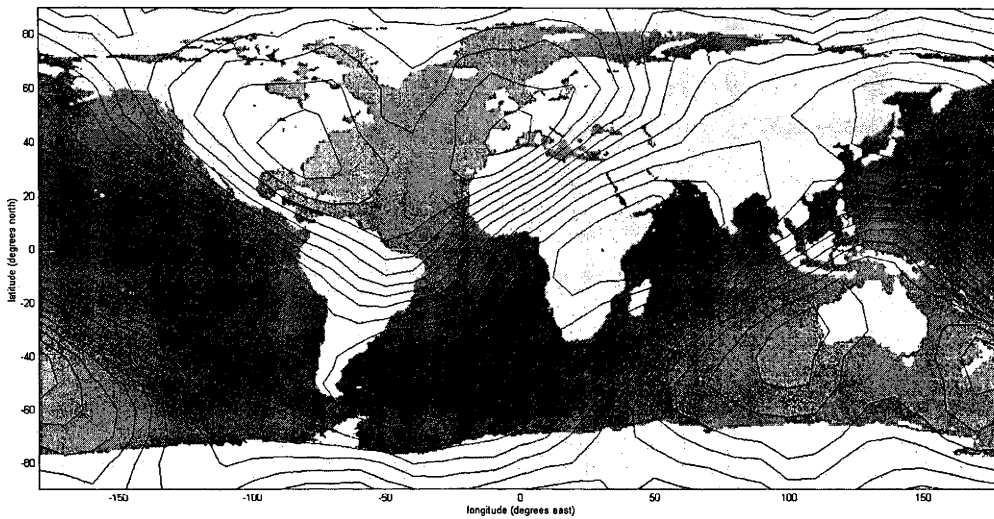


Figure 5.3: Relative merit of an additional site to augment a network consisting of the LIGO Hanford (4km) Observatory, the LIGO Livingston Observatory and a 4km LIGO I instrument at the VIRGO site, in a coincident analysis (lighter is better, contours every 2.5%). The minimum detection rate is 69% of the maximum.

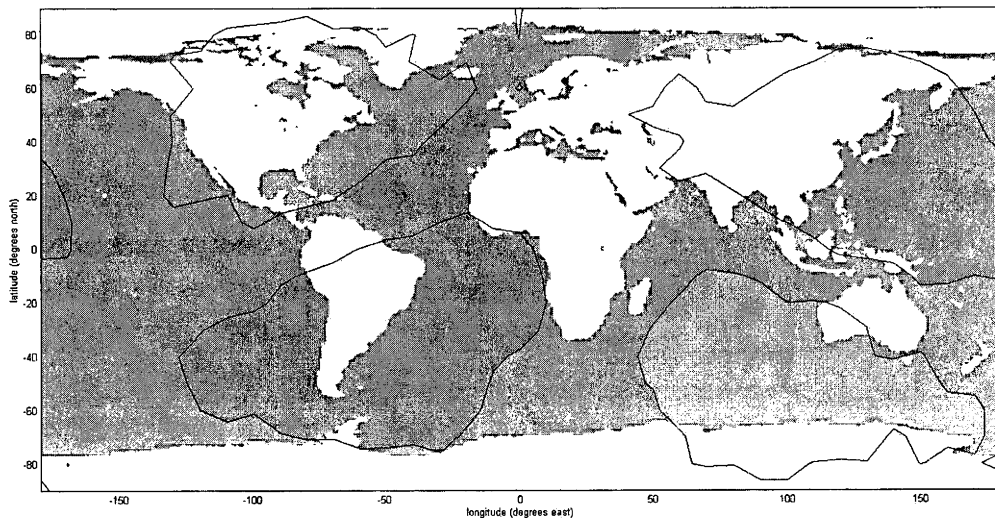


Figure 5.4: Relative merit of an additional site to augment a network consisting of the LIGO Hanford (4km) Observatory, the LIGO Livingston Observatory and a 4km LIGO I instrument at the VIRGO site, in a coherent analysis (lighter is better, contours every 2.5%). The minimum detection rate is 94% of the maximum.

the network varies moderately with location, with multiple minima of about 70% of the best achievable detection rates.

Under a coherent network analysis in Figure 5.4, we once again see a qualitative similarity to Figure 5.3 in the locations of maxima and minima, but quantitatively much less variation than in the coincident case, with only 6% separating the best and worst sites. As expected, this indicates that AIGO is an optimal site to augment the existing global network; however, the weak dependence of event rate on geographical location for a coherent analysis means that its advantage over other sites is slight.

5.3 Conclusion

We have proposed a formalism for conducting studies of the relative merits of differently configured systems of gravitational wave observatories and different collaborative analysis techniques.

We have demonstrated that, given certain assumptions, simple imple-

mentations of the coincident and coherent analysis techniques exhibit very different dependencies on the geographical locations of their component detectors.

Under our model, it is clear that the (binary inspiral) detection rate for a global network is insensitive to the geographical configuration of its component detectors when a coherent analysis is used, in contrast to when a simple coincident analysis is used. Whilst the LIGO detectors and the proposed AIGO detector are well sited to complement one another under a coincident analysis, the sites of the VIRGO detector and the proposed LCGT detector are far from optimal; our results demonstrate that under a coherent analysis the cost of this sub-optimal siting is substantially reduced, on at least one figure of merit. In this sense, the global network is closer to optimal for a coherent analysis than for a coincident analysis. Our results also indicate that since, under a coherent analysis, detection rate is insensitive to detector siting, the location of an augmenting detector could be optimised for other network properties (for example, directional resolution) without compromising the event rate.

It is important to note that the model does not compare the absolute detection rates for the two analysis techniques; we cannot say that one method would produce a higher detection rate than the other for a given false alarm rate. Though we have considered only one class of source and one figure of merit, our formalism is general enough to extend to more general problems.

Chapter 6

Continuous-wave sources

6.1 Introduction

The most optimistic estimates place the strain produced by continuous wave sources at least three orders of magnitude below those of inspiral and other burst events [34]. Hopes of their detection are due to the fact that the signal can be integrated over months, or potentially even years of observation.

The motions of the Earth serve to modulate the incoming continuous gravitational wave signal. Daily rotation varies the angle of the source, and hence the sensitivity of the detector, leading to variations in amplitude. Orbital motions provide a seasonally-varying Doppler shift to the frequency.

Currently, it is computationally unfeasible to cover all the possible parameters governing the waveform (as is done with template banks for inspirals). Optimal searches are restricted to match the parameters of nearby pulsars known from their electromagnetic emissions [34]. Hierarchical searches, computationally feasible but with less-than-optimal sensitivity, will be used to search for continuous wave sources not associated with electromagnetically identified pulsars.

6.2 Methodology

Consider a neutron star and an interferometric gravitational wave detector. The neutron star has principal axes I_1 , I_2 and I_3 , and rotates about I_3 with angular frequency ω_p . The neutron star is a distance r from the detector, and I_3 is inclined at an angle i to the line of sight is oriented at an angle ψ_p

from geographic north. The strains produced for the two polarisations along the line of sight are

$$h_+(t) = \frac{\omega_g^2 I \varepsilon}{r} \left(\frac{1 + \cos^2 i}{2} \right) \cos \omega_g t \quad (6.1)$$

$$h_\times(t) = \frac{\omega_g^2 I \varepsilon}{r} \cos i \sin \omega_g t \quad (6.2)$$

where

$$I = \frac{I_1 + I_2}{2} \quad (6.3)$$

$$\varepsilon = \frac{I_1 - I_2}{I} \quad (6.4)$$

$$\omega_g = 2\omega_p, \quad (6.5)$$

noting that the frequency of the gravitational waves ω_g is twice that of the pulsar. The strain measured by the observatory will be

$$h(t) = F_+ h_+(t) + F_\times h_\times(t) \quad (6.6)$$

where the antenna-pattern factors F_+ and F_\times are functions of the (time-varying) relative orientation of the neutron star and the detector.

$$h(t) = \frac{\omega_g^2 I \varepsilon}{r} \left[F_+ \left(\frac{1 + \cos^2 i}{2} \right) \cos \omega_g t + F_\times \cos i \sin \omega_g t \right] \quad (6.7)$$

This is a sinusoid of some amplitude A and phase β

$$h(t) = A \cos(\omega_g t + \beta) \quad (6.8)$$

where

$$A^2 = \left(\frac{\omega_g^2 I \varepsilon}{r} \right)^2 \left[F_+^2 \frac{(1 + \cos^2 i)^2}{4} + F_\times^2 \cos^2 i \right]. \quad (6.9)$$

We neglect motions of the Earth (and indeed the neutron star) other than their rotation.

As the Earth, and any ground-based interferometer, rotates once each sidereal day, the value of A^2 will vary with this period, T . Moreover, this

value will not depend on the right ascension ϕ_p of the neutron star. The average over one sidereal day is equal to the average over right ascension,

$$\overline{A^2} = \left(\frac{\omega_g^2 I \varepsilon}{r} \right)^2 \frac{1}{2\pi} \int_{-\pi}^{\pi} F_+^2 \frac{(1 + \cos^2 i)^2}{4} + F_-^2 \cos^2 i \, d\phi_p \quad (6.10)$$

where the parameters of the observatory (most importantly, its latitude and orientation) are implicit in the antenna patterns.

6.2.1 Implementation

The solution of the integral in 6.10 is a simple but arduous process. Noting that

$$\hat{\mathbf{e}}_x = \begin{bmatrix} -\sin \theta \cos \phi \cos \psi - \sin \phi \sin \psi \\ -\sin \theta \sin \phi \cos \psi + \cos \phi \sin \psi \\ \cos \theta \cos \psi \end{bmatrix} \quad (6.11)$$

$$\hat{\mathbf{e}}_y = \begin{bmatrix} -\sin \theta \cos \phi \sin \psi + \sin \phi \cos \psi \\ -\sin \theta \sin \phi \sin \psi - \cos \phi \cos \psi \\ \cos \theta \sin \psi \end{bmatrix} \quad (6.12)$$

and from Equation 4.5,

$$\mathbf{R} = \hat{\mathbf{e}}_x \hat{\mathbf{e}}_x^T - \hat{\mathbf{e}}_y \hat{\mathbf{e}}_y^T, \quad (6.13)$$

we may expand the dependance of \mathbf{R} on ϕ as follows:

$$\mathbf{R}(\phi) = \mathbf{R}_a \cos^2 \phi + \mathbf{R}_b \sin^2 \phi + \mathbf{R}_c \cos \phi \sin \phi + \mathbf{R}_d \cos \phi + \mathbf{R}_e \sin \phi + \mathbf{R}_f, \quad (6.14)$$

where

$$\mathbf{R}_a = \begin{bmatrix} \sin^2 \theta \cos 2\psi & -\sin \theta \sin 2\psi & 0 \\ -\sin \theta \sin 2\psi & -\cos 2\psi & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (6.15)$$

$$\mathbf{R}_b = \begin{bmatrix} -\cos 2\psi & 0 & \sin \theta \sin 2\psi \\ \sin \theta \sin 2\psi & \sin^2 \theta \cos 2\psi & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (6.16)$$

$$\mathbf{R}_c = \begin{bmatrix} 2 \sin \theta \sin 2\psi & (\sin^2 \theta + 1) \cos 2\psi & 0 \\ (\sin^2 \theta + 1) \cos 2\psi & -2 \sin \theta \sin 2\psi & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (6.17)$$

$$\mathbf{R}_d = \cos \theta \begin{bmatrix} 0 & 0 & \sin \theta \cos 2\psi \\ 0 & 0 & \sin 2\psi \\ \sin \theta \cos 2\psi & \sin 2\psi & 0 \end{bmatrix} \quad (6.18)$$

$$\mathbf{R}_e = \cos \theta \begin{bmatrix} 0 & 0 & -\sin 2\psi \\ 0 & 0 & -\sin \theta \cos 2\psi \\ -\sin 2\psi & -\sin \theta \cos 2\psi & 0 \end{bmatrix} \quad (6.19)$$

$$\mathbf{R}_f = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \cos^2 \theta \cos 2\psi \end{bmatrix} \quad (6.20)$$

The sidereally averaged squared response—the incident power—is given by

$$\begin{aligned} & \int_{-\pi}^{\pi} \left(\sum_{i,j=1}^3 R_{ij}(\phi) H_{ij} \right)^2 d\phi \\ &= \int_{-\pi}^{\pi} (S_a \cos^2 \phi + S_b \sin^2 \phi + S_c \cos \phi \sin \phi + S_d \cos \phi + S_e \sin \phi + S_f)^2 d\phi \\ &= \frac{\pi}{4} (3S_a^2 + 3S_b^2 + S_c^2 + 4S_d^2 + 4S_e^2 + 8S_f^2 + 2S_a S_b + 8S_a S_f + 8S_b S_f), \end{aligned} \quad (6.21)$$

where

$$S_a = \sum_{i,j=1}^3 (\mathbf{R}_a)_{ij} H_{ij}, \quad (6.22)$$

and similarly for S_b *et cetera*.

The sidereally-averaged squared response of a detector to a strain is readily computed in MATLAB; see Appendix B.

The sidereally-averaged antenna patterns are depicted in Figure 6.1. Notably, those of observatories at latitudes $\pm \frac{\pi}{2}$ retain their characteristic “peanut” shape, but their dimpled minima are averaged out. In contrast, at near-0 (equatorial) latitudes, the primary lobes have been swept by the average into a torus-like structure whose exact shape depends on the orientation of the detector. Note that an equatorial detector with a $\frac{\pi}{4}$ orientation is completely insensitive to sources at the celestial poles. Existing detectors in the $\frac{\pi}{6}$ to $\frac{\pi}{3}$ latitudes have intermediate forms.

6.3 Detection

For a single detector and a particular neutron star, the neutron star is detectable if the mean square amplitude exceeds a certain threshold Λ^2 . For

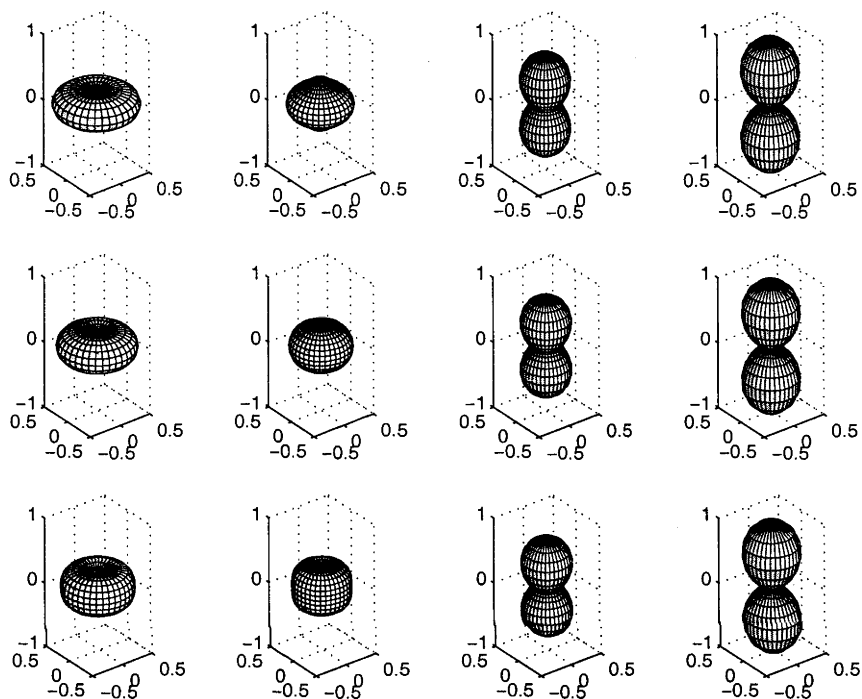


Figure 6.1: Sidereally-averaged response to a uniform distribution of pulsars of interferometers with varying latitudes and orientations—effectively the familiar peanut antenna pattern averaged over a rotation. The responses are independent of longitude; the vertical axis of the diagram is the Earth’s axis of rotation. From left to right, latitudes of 0° , $\pm 30^\circ$, $\pm 60^\circ$ and $\pm 90^\circ$. From top to bottom, orientations of 0 , $\frac{\pi}{8}$ and $\frac{\pi}{4}$ from north. Note that for the equatorial detector with a $\frac{\pi}{4}$ orientation, an antenna pattern null aligns with the Earth’s axis of rotation so that no sources from that direction could be detected.

a given declination, orientation and inclination, the maximum distance r_{\max} that such a neutron star can be detected is

$$\Lambda^2 = \overline{A^2} \quad (6.23)$$

$$= \left(\frac{\omega_g^2 I \varepsilon}{r_{\max}} \right)^2 \frac{1}{2\pi} \int_{-\pi}^{\pi} F_+^2 \frac{(1 + \cos^2 i)^2}{4} + F_{\times}^2 \cos^2 i \, d\phi_p \quad (6.24)$$

$$r_{\max}^2 = \left(\frac{\omega_g^2 I \varepsilon}{\Lambda} \right)^2 \frac{1}{2\pi} \int_{-\pi}^{\pi} F_+^2 \frac{(1 + \cos^2 i)^2}{4} + F_{\times}^2 \cos^2 i \, d\phi_p \quad (6.25)$$

Under the simplification that this threshold is the same for all observable neutron star parameters, we would like to equate this *seeing distance* r_{\max} with an observable volume of space. To do this we must, however, form not an average seeing distance, but rather the average cubed seeing distance $\overline{r_{\max}^3}$ over the different seeing distances for different orientations ψ_p and inclinations i_p of the population of neutron stars along the line of sight.

$$\overline{r_{\max}^3} = \frac{1}{2\pi} \int_{-\pi/2}^{\pi/2} \frac{1}{2} \int_{-\pi}^{\pi} r_{\max}^3 \cos i \, di_p d\psi_p \quad (6.26)$$

so that the total observable volume is

$$V = 2\pi \int_{-\pi/2}^{\pi/2} \overline{r_{\max}^3} \cos \theta_p \, d\theta_p, \quad (6.27)$$

noting that r_{\max} is independent of right ascension.

6.4 Galactic distribution

A simple model of neutron star distribution in the galaxy (Figure 6.2) is provided by [35]. For a population of neutron stars with a particular set of intrinsic parameters, we can determine what proportion of the total galactic population may be detected by a particular observatory for a particular threshold.

The result for any given threshold may be computed as a Monte-Carlo integration, but the same computation can be used to simultaneously compute the fraction for any threshold. We store all the thresholds computed in the Monte-Carlo sum, and sort it into a monotonically-decreasing list $(\overline{A^2})_i$.

Galactic neutron star population

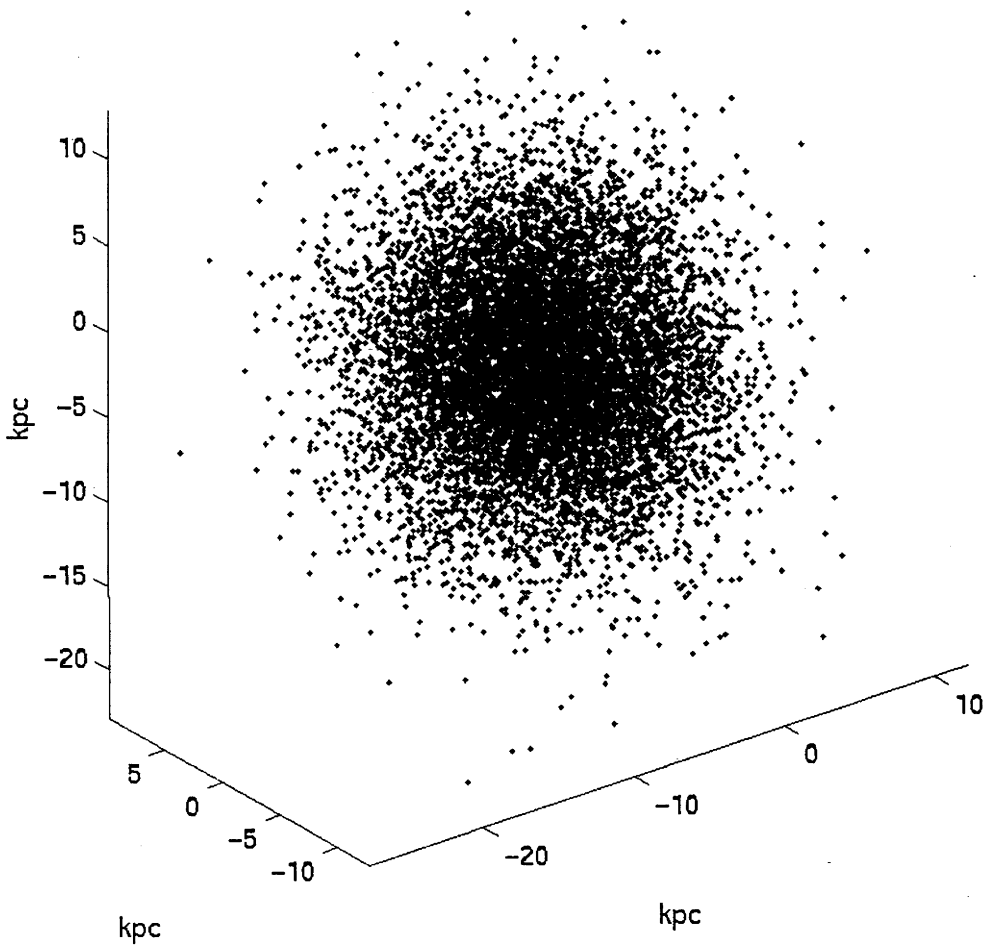


Figure 6.2: Model for the distribution of galactic pulsars, in celestial coordinates.

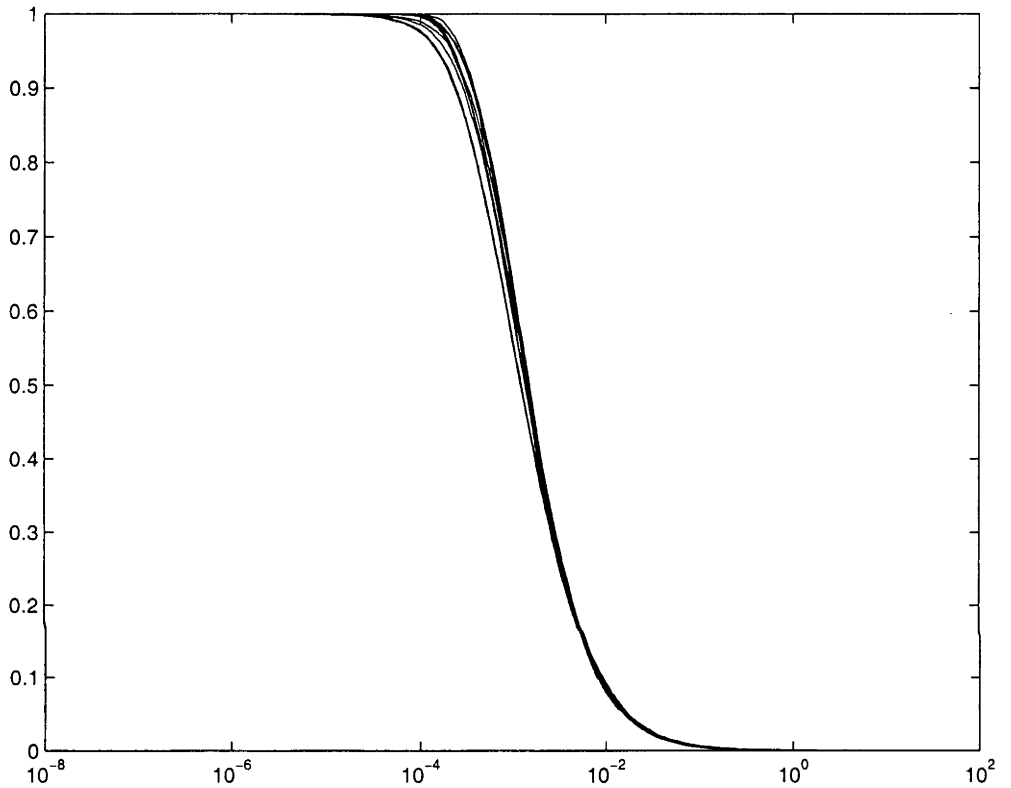


Figure 6.3: Detectable fraction (vertical) of a galactic pulsar population against relative detection (horizontal) threshold for various detector latitudes and orientations (lines). The latitude and orientation have a minimal effect on an detector's ability to observe galactic neutron stars, which is almost wholly governed by its baseline strain sensitivity.

For a particular threshold $(\overline{A^2})_i$, its normalised position in the list i/N is the detectable population fraction.

Figure 6.3 shows this relation for a variety of detector latitudes and orientations. As we would expect, in the regimes of very high and very low sensitivity, the position of the detectors is irrelevant. However, even at intermediate sensitivities, the siting of an observatory has only a minimal impact. Strain sensitivity, not geometry, dominates.

6.5 Conclusion

Despite the large differences in the sidereally-averaged antenna-patterns of terrestrial gravitational wave observatories at different latitudes, the fraction of a galactic population of neutron stars they can detect is almost independent of latitude. (Of course, their ability to detect any particular neutron star is highly dependent on latitude.)

Chapter 7

Summary and future directions

The practical scientist is trying to solve tomorrow's problems with yesterday's computer

— Numerical Recipes

7.1 Data conditioning

... and perhaps tomorrow's as-yet-unfinished C++ compiler.

LDAS is quite an achievement, engineered by only a handful of hardy souls and now entrenched at the very heart of LIGO science (Chapter 2). As with any complicated project, hindsight suggests alternative routes and useful lessons, but the most important fact to note is that LDAS delivered the extensive, and extensively tested, functionality that the S1 analyses [14, 34, 36, 37] required.

The data conditioning API is the bridge between LDAS's roles as LIGO's data librarian and search coordinator, taking raw data and converting it into the input expected by each particular search code. The author participated in the development and implementation of the Universal Data Type framework that underpins the current data conditioning API, allowing the uniform support of the API's many different types of data—scalars, vectors, and matrices of integer, real or complex values represented at various precisions. He also participated in the design, implementation and testing of the basic signal processing operations—heterodyning, linear filtering and resampling—and their integration into the data conditioning API command language, as well as in the extensive testing including Mock Data Challenges.

The line removal algorithm built using that functionality promises to be a useful tool for removing environmental disturbances [15]. It has been extensively characterised, leading to its inclusion in the first stochastic background upper limits analysis. Rather than conditioning data for the analysis, it was used to prove that the then level of correlations introduced by spectral lines did not have a significant impact on the analysis. Without this result there may have been less confidence in the stochastic background upper limit results [14].

OELSLR compares favourably with earlier line removal techniques, in terms of its ability to recover signals from beneath lines without attenuation and because of its ready availability—it can be deployed for any search code merely by pasting a few lines of text into the search’s `ldasJob`. Though currently only applied to the anthropogenic 60 Hz lines, the model itself is exceedingly general and could be applied to any measurable environmental disturbance over narrow or broad frequency bands. Obvious candidates are low-frequency seismic noise, vibrational and acoustic couplings, and magnetic effects other than the 60 Hz lines—in fact most of the physical environment monitor channels. Further work will identify the optimal parameters for these new operating regimes, and perhaps a way of automatically identifying the parameters. Application to searches beyond the stochastic background will either improve their sensitivity or prove their resilience against environmentally induced correlations.

Generalisation of the underlying system identification model could only improve the performance of the line remover. Allowing multiple input channels could permit the system to remove many kinds of environmental noise simultaneously. The system identification model could be improved from an auto-regression to an output error or even more advanced system, though this would be a major undertaking. Finally, the model could be made adaptive, so that it ‘automagically’ refines its model as it processes data.

The future of LDAS itself is ultimately in the hands of its users. So far it has seen great use as a data distribution technology throughout LIGO, and as one of a variety of analysis tools employed by science searches. In the future it is likely to migrate to data grid and even computation grid technologies, linking together clusters across the US (and Australia) into coherent virtual machines.

7.2 Network simulation

The interplay between a gravitational wave observatory's location, orientation and sensitivity is intriguing. Most sensitive to gravitational wave sources directly above or below the plane of the two (perpendicular) arms, there is lesser interferometer sensitivity to sources from almost all other directions, so phase and timing information must be used to reconstruct the direction and polarisation of sources from the output of multiple interferometers. Useful time delays between interferometers correspond to many hundreds of kilometres of separation, and as the interferometers are constrained to lie on the curved surface of the Earth, this produces a misalignment of the instruments. The exception is when detectors are situated on opposite sides of the Earth, as would be the case for an Australian complement to the North American detectors.

We first considered the detection of known short waveforms—the classic inspiral source, though our model is slightly more widely applicable—by two different techniques: the simple coincidence test (currently implemented by the global network of bar detectors), and a fully coherent search (analogous to aperture synthesis in radio astronomy). By varying the configuration of one interferometer while keeping the others—corresponding to existing observatories—fixed, and performing a Monte-Carlo estimate of the sensitivity of the whole network, we were able to plot out the relative merit of new observatory locations. Western Australia, by virtue of its antipodean location, was always an optimal location. The difference between the best and worst configurations for a coherent search was only a few percent suggesting that a coherent search is more robust against misalignment and thus more likely to suit a real global network. When considering the detection of continuous wave sources—specifically a galactic population of neutron stars—only the faintest of dependencies on detector configuration was observed. These results are good news for global gravitational wave astronomy: the geographical arrangement of existing gravitational wave interferometers will not significantly impair their ability to work together as a single global instrument.

The models used are plausible, but for computational tractability a large number of simplifications had to be made—most importantly the assumptions of identical interferometers and neglecting issues of frequency response and the impact of environmental noise. The LDAS implementation on the growing ACIGA Data Analysis Cluster gives us the computing horsepower and framework to perform much more in-depth and realistic studies, even

up to testing the recovery of injected signals by mocked-up network search codes based on their real single-detector equivalents. Using the ACIGA physical environment monitoring station at The Australian National University and data from our overseas partners, we will be able to simulate a southern hemisphere detector with the characteristics of a LIGO instrument, yet with southern hemisphere environmental disturbances and a southern hemisphere response time and matrix to simulated signals.

Perhaps the most promising role of an Australian detector is not so much to add to the detection capability of a network, but rather by providing a long baseline to the mid-latitudes of the northern hemisphere, and improving the angular resolution of the whole network—a study which is a priority of ACIGA. With an application made for funds in 2005 to model an Australian detector, including angular resolution and global noise correlations, we may soon have definitive answers.

7.3 Conclusion

My development of aspects of the LDAS Data Conditioning API comprised part of the infrastructure for the LIGO upper limits papers, [14, 34, 36, 37], some of the first ‘big science’ results of the 21st century. I also considered how to optimally develop the astronomical discovery capacity of a global network of gravitational wave observatories, particularly focussing on the role Australia might play.

I am proud to continue to play my small part in this great scientific adventure.

— Antony Charles Searle, Canberra, 2004

Appendix A

Line remover implementation

Implementation is naturally broken into two pieces of functionality: an output-error model, and a band-selector. They are tied together by a line-remover class.

Disclaimer The code presented here is substantially, but not entirely, the work of the author. Contributions of others are limited to stylistic issues and minor compatibility issues.

A.1 Band selection

Though seemingly trivial, the combination of heterodyning, resampling, and the introduced transients and delays made band selection a difficult task to get right. Eventually, it was abstracted into a class.

To select a band centred around f of width f_{Ny}/n , and then to reverse that selection, it is necessary to create two Mixer objects, mixing by $\pm f/f_{Ny}$, and two Reasample objects, up- and down-sampling by n .

Listing A.1: BandSelector class definition.

```
class BandSelector
{
public:
    BandSelector(const double& frequency, const std::size_t factor)
```

```

        : m_downmixer(MixerState(0.0, -frequency))
        , m_downsampler(1, factor)
        , m_downshifter(0)
        , m_upsampler(factor, 1)
        , m_upshifter(0)
        , m_upmixer(MixerState(0.0, +frequency))
    {
    }

BandSelector(const BandSelector& bs)
    : m_downmixer(bs.m_downmixer)
    , m_downsampler(bs.m_downsampler)
    , m_downshifter(bs.m_downshifter ? bs.m_downshifter->Clone
        () : 0)
    , m_upsampler(bs.m_upsampler)
    , m_upshifter(bs.m_upshifter ? bs.m_upshifter->Clone() : 0)
    , m_upmixer(bs.m_upmixer)
    {
    }

~BandSelector()
{
    delete m_downshifter;
    delete m_upshifter;
}

BandSelector& operator=(const BandSelector& bs)
{
    if (&bs != this)
    {
        m_downmixer = bs.m_downmixer;
        m_downsampler = bs.m_downsampler;
        delete m_downshifter;
        m_downshifter = bs.m_downshifter ? bs.m_downshifter->
            Clone() : 0;
        m_upsampler = bs.m_upsampler;
        delete m_upshifter;
        m_upshifter = bs.m_upshifter ? bs.m_upshifter->Clone()
    }
}

```

```

        : 0;
        m_upmixer = bs.m_upmixer;
    }
    return *this;
}

BandSelector* clone() const
{
    return new BandSelector(*this);
}

template<typename out_, typename in_>
void apply(std::valarray<out_>& out, const std::valarray<in_>& in)
{
    std::valarray<out_> downmixed;
    m_downmixer.apply(downmixed, in);
    Sequence<out_> downsampled;
    m_downsampler.apply(downsampled, downmixed);
    if (!m_downshifter) m_downshifter = new ShiftState<out_>(
        m_downsampler.getDelay(), 1);
    dynamic_cast<ShiftState<out_>&>(*m_downshifter).apply(
        downsampled);
    out.resize(downsampled.size());
    out = downsampled;
}

template<typename out_, typename in_>
void ylppa(std::valarray<out_>& out, const std::valarray<in_>& in)
{
    Sequence<in_> upsampled;
    m_upsampler.apply(upsampled, in);
    if (!m_upshifter) m_upshifter = new ShiftState<in_>(
        m_upsampler.getDelay(), 1);
    dynamic_cast<ShiftState<in_>&>(*m_upshifter).apply(
        upsampled);
    m_upmixer.apply(out, upsampled);
}

```

```

    }

private:

    BandSelector();

    Mixer m_downmixer;
    Resample m_downsampler;
    State* m_downshifter;
    Resample m_upsampler;
    State* m_upshifter;
    Mixer m_upmixer;

};

```

A.2 Output-error model

The development of components for the line remover occurred when the Data Conditioning API was comparatively complete and stable. Knowledge of the issues encountered in the development of earlier components allowed a different approach to be taken in their implementation.

The output-error model exposes only one class to the user, which both stores internal state and applies the action. As such, the `OEModel` class inherits from `State`.

The type of the internal data is determined by the type of the series the model is required to estimate; all subsequent methods must be invoked with compatible types. Methods that invalidate the estimated model can be used to reset the instance to an uninitialised state.

Listing A.2: `OEModel` interface.

```

class OEModel :
public State
{
public:
    virtual ~OEModel();
    OEModel* Clone() const;

```

A broad range of constructors are supplied, supporting initialisation from another OEModel, from the model orders n_b and n_f , from a raw model state θ , or to automatically model a given system. Where appropriate, the constructors are templatised and given UDT parallels.

Listing A.3: OEModel interface (continued).

```
OEModel(); // blank model
OEModel(const OEModel&);
OEModel(const int& order_b);
OEModel(const int& order_b, const int& order_f);
template<typename type> explicit OEModel(const std::valarray<
    type>& theta, const int& order_b);
explicit OEModel(const UDT& order_b);
OEModel(const UDT& order_b_or_theta, const UDT&
    order_f_or_order_b);
template<typename type>
    OEModel(const std::valarray<type>& y, const std::valarray<type
        >& u, const int& order_b);
template<typename type>
    OEModel(const std::valarray<type>& y, const std::valarray<type
        >& u, const int& order_b, const int& order_f);

OEModel(const UDT& y, const UDT& u, const UDT& order_b);
OEModel(const UDT& y, const UDT& u, const UDT& order_b,
    const UDT& order_f);
```

A large number of accessors are provided for the data stored by the model. Those which write to a provided argument require the correct types be provided, or an exception will be thrown.

A model is created in an undefined state. The model order is then provided. Finally the series u and y are provided and a model for y in terms of u is estimated. The model may be applied to different epochs of u to produce predictions of the corresponding y , or if u and y are provided for a different epoch, the model may be further refined.

Listing A.4: OEModel interface (continued).

```
// accessors
int getOrderB() const;
int getOrderF() const;
```

```

void getOrderB(UDT*& order_b) const;
void getOrderF(UDT*& order_f) const;
template<typename type>
    void getTheta(std::valarray<type>& theta) const;
void getTheta(UDT*& theta) const;
void getFilterB(UDT*& filter_b) const;
void getFilterF(UDT*& filter_f) const;
template<typename type>
    void getFilterB(std::valarray<type>&) const;
template<typename type>
    void getFilterF(std::valarray<type>&) const;

```

Mutators are also provided. They typically reset the internal state.

Listing A.5: OEModel interface (continued).

```

OEModel& operator=(const OEModel&);
void setOrderB(const int& order_b);
void setOrderF(const int& order_f);
void setOrderB(const UDT& order_b);
void setOrderF(const UDT& order_f);
template<typename type>
    void setTheta(const std::valarray<type>& theta, const int&
        order_b);
void setTheta(const UDT& theta, const UDT& order_b);

```

As well as apply methods that accept a series u and return an estimate of y , the refine methods can use additional u and y series to improve the model.

Listing A.6: OEModel interface (continued).

```

template<typename type>
    void apply(std::valarray<type>& w,
        const std::valarray<type>& u) const;
void apply(UDT*& w,
    const UDT& u) const;
template<typename type>
    void refine(const std::valarray<type>& y, const std::valarray<type>
        & u);
void refine(const UDT& y, const UDT& u);

```

A figure of merit for the model's performance can be computed, and may be used to automatically estimate a good model order for the system.

Listing A.7: OEModel interface (continued).

```

template<typename type>
    type merit(const std::valarray<type>& y, const std::valarray<type>
        & u) const;
    void merit(UDT*& m, const UDT& y, const UDT& u) const;

```

The way OEModel stores its internal state is instructive.

Listing A.8: OEModel interface (continued).

```

private:
    int m_order_b;
    int m_order_f;
    class Abstraction;
    template<typename type> class Implementation;
    mutable Abstraction* m_data;
};

```

While data common to all input data precisions—the model orders—is stored trivially, the rest of the state can be real or complex, single or double precision, depending on the nature of the input. OEModel stores a pointer to an implementation of the state, accessed through as the abstract base class of the implementations for various precisions. This model has the benefit of allowing **virtual** functions to perform some of the odious type-checking tasks caused by the UDT.

Listing A.9: OEModel state abstraction.

```

class OEModel::Abstraction
{
    public:
        virtual ~Abstraction() = 0; // abstract base class
        virtual Abstraction* clone() const = 0; // enable copy without
            knowledge of exact type, like a UDT
        virtual void getTheta(UDT*&) const = 0; // accessors
        virtual void getFilterB(UDT*&) const = 0;
        virtual void getFilterF(UDT*&) const = 0;
        virtual void apply(UDT*& w, const UDT& u) const = 0;
        virtual void refine(const UDT& y, const UDT& u) = 0;
        virtual void merit(UDT*& m, const UDT& y, const UDT& u) const
            = 0;

```

```

protected: // only derived classes can perform the following
    Abstraction();
    Abstraction(const Abstraction&);
    Abstraction& operator=(const Abstraction&);
};

```

Abstraction passes on all the accessors and mutators involving UDTs and hence type resolution via virtual functions; their overrides in types derived from Abstraction will support exactly one UDT type and can avoid switch-on-type blocks.

Abstraction does not support methods that are statically typed; these are only valid for a single Implementation type. This means that Implementation has a larger interface than Abstraction. It also stores the actual model data. Implementation in fact resembles OEModel—except that the class itself is templatised, not its methods, and the methods accept the model orders as arguments, passed to them from OEModel.

Listing A.10: OEModel state implementation.

```

template<typename type> class OEModel::Implementation :
    public OEModel::Abstraction
{
public:
    Implementation();
    Implementation(const Implementation&);
    virtual ~Implementation();
    Implementation& operator=(const Implementation&);
    virtual Implementation* clone() const;
    explicit Implementation(const int& order_b, const int& order_f =
        order_b);
    explicit Implementation(const std::valarray<type>& theta, const int&
        order_b);
    Implementation(const std::valarray<type>& y, const std::valarray<type>
        & u, const int& order_b, const int& order_f);
    virtual void getTheta(UDT*&) const;
    virtual void getFilterB(UDT*&) const;
    virtual void getFilterF(UDT*&) const;
    void getTheta(valarray<type>&) const;
    void getFilterB(valarray<type>&) const;
    void getFilterF(valarray<type>&) const;

```



```

void apply(valarray<type>& w, const valarray<type>& u) const;
void apply(UDT*& w, const UDT& u) const;
void refine(const valarray<type>& y, const valarray<type>& u);
void refine(const UDT& y, const UDT& u);
type merit(const valarray<type>& y, const valarray<type>& u) const;
void merit(UDT*& m, const UDT& y, const UDT& u) const;

```

The data stored by the implementation serves several purposes. The model itself is stored in `m_theta`. `m_matrix` and `m_vector` are the raw products of the estimation method, allowing an estimate to be continued and refined. `m_state` is the state of the custom linear filter invoked by `OEModel`. (`LinFilt` supported only real coefficients at the time of development.)

Listing A.11: `OEModel` state implementation (continued).

```

private:
    valarray<type> m_theta;
    mutable valarray<type> m_state;
    // !!! ARX estimator *not* OE estimator !!
    Matrix<type> m_matrix;
    Matrix<type> m_vector;
    std::valarray<type> m_history_y;
    std::valarray<type> m_history_u;
};

```

Several of the implementations of `OEModel`'s methods are instructive.

Listing A.12: Progressive model estimator.

```

template<typename type> void Implementation<type>::refine(const
    valarray<type>& y, const valarray<type>& u)
{
    // Solve  $Ax = b$  for  $x$  where
    //  $A = \sum_{t=1}^N \psi(t)\psi^T(t)$ 
    //  $b = \sum_{t=1}^N \overline{\psi(t)}y(t)$ 
    // and
    //  $\psi(t) = [u(t-1) \dots u(t-n_b) \ y(t-1) \dots y(t-n_f)]^T$ 
    valarray<type> intermediate(m_theta.size());
    valarray<type> temporary(m_theta.size());
    for (unsigned int t = 0; t < u.size(); ++t)
    {

```

```

intermediate[std::slice(0, m_history_u.size(), 1)] = m_history_u;
intermediate[std::slice(m_history_u.size(), m_history_y.size(), 1)] =
    m_history_y;
temporary = conj(intermediate);
// accumulate A
for (unsigned int i = 0; i < m_theta.size(); ++i)
    m_matrix.column(i) += (temporary * intermediate[i]);

m_vector.column(0) += (temporary * y[t]);

// rotate psi

if (m_history_u.size() > 0)
{
    m_history_u = m_history_u.shift(-1);
    m_history_u[0] = u[t];
}
if (m_history_y.size() > 0)
{
    m_history_y = m_history_y.shift(-1);
    m_history_y[0] = y[t];
}
}

Matrix<type> buffer;
TheCLAPACKSoHandle.SV(m_matrix, buffer, m_vector);
m_theta = buffer.column(0);
}

```

Listing A.13: OEModel linear filter implementation.

```

template<typename type> void Implementation<type>::apply(valarray
<type>& w, const valarray<type>& u) const
{
    w.resize(u.size());
    if (m_theta.size() > 0)
    {
        for (unsigned int t = 0; t < u.size(); ++t)
        {

```

```

        w[t] = m_state[0];
        m_state = m_state.shift(1);
        m_state[std::slice(0, m_history_u.size(), 1)]
            += m_theta[std::slice(0, m_history_u.size(), 1)] * u[t];
        m_state[std::slice(0, m_history_y.size(), 1)]
            += m_theta[std::slice(m_history_u.size(),
                                m_history_y.size(), 1)] * w[t];
    }
}
}

```

A.3 Interface

Listing A.14: LineRemover interface.

```

#ifndef LINE_REMOVER_HH
#define LINE_REMOVER_HH

#include <complex>
#include <valarray>
#include <vector>

#include "StateUDT.hh"

namespace datacondAPI
{

    class LineRemover : public State
    {
    public:

        // override defaults

        //: Construct a line remover
        LineRemover(const double& frequency,
                   const std::size_t& factor,

```

```

        const std::size_t& order);

    ///  

    LineRemover(const UDT& frequency, const UDT& factor, const  

        UDT& order);

    ///  

    ///  

    LineRemover(const LineRemover& a);

    ///  

    ~LineRemover();

    ///  

    LineRemover& operator=(const LineRemover& a);

    ///  

    // UDT functionality

    ///  

    virtual LineRemover* Clone() const;

    virtual ILwd::LdasElement*  

        ConvertTollwd(const CallChain& Chain,  

            UDT::target_type Target = UDT::TARGET_GENERIC)  

            const;

    ///  

    void getFilter(UDT*& b) const;

    ///  

    // fit model

    ///  

    ///  

    ///  

    template<typename type>  

    void refine(const std::valarray<type>& y,

```



```

    };
}

#endif // LINE_REMOVER_HH

```

Listing A.15: LineRemover implementation.

```

#include <algorithm>
#include <climits> //:todo: <limits>

#include <general/unimplemented_error.hh>

#include "fft.hh"
#include "ifft.hh"
#include "LineRemover.hh"
#include "OEModel.hh"
#include "ScalarUDT.hh"
#include "SequenceUDT.hh"
#include "Mixer.hh"
#include "Resample.hh"
#include "ShiftState.hh"

namespace datacondAPI
{
    template<typename T> struct complex_traits
    {
        typedef std::complex<T> complex_type;
        typedef T real_type;
    };

    template<typename T> struct complex_traits<std::complex<T> >
    {
        typedef typename complex_traits<T>::complex_type
            complex_type;
        typedef typename complex_traits<T>::real_type real_type;
    };
}

```

```
class LineRemover::Abstraction
{
public:
    // override defaults
    virtual ~Abstraction();

    // interface
    virtual Abstraction* clone() const = 0;
    virtual void apply(UDT*& w, const UDT& u) = 0;
    virtual void getFilter(UDT*& b) const = 0;

protected:
    Abstraction();
    Abstraction(const Abstraction&);

    Abstraction& operator=(const Abstraction&);

private:
};

template<typename type>
class LineRemover::Implementation : public Abstraction
{
public:
    // override defaults

    Implementation();
    Implementation(const Implementation&);
```

```

virtual ~Implementation();

Implementation& operator=(const Implementation&);

// interface

virtual Implementation* clone() const;

void refine(const std::valarray<type>& y,
            const std::valarray<type>& u,
            const double& frequency,
            const std::size_t& factor,
            const std::size_t& order);

virtual void apply(UDT*& w,
                  const UDT& u);

void apply(std::valarray<type>& w,
            const std::valarray<type>& u);

virtual void getFilter(UDT*& b) const;

private:

    OEMModel* m_model;

    BandSelector* m_refine_y;
    BandSelector* m_refine_u;

    BandSelector* m_apply_y;
    BandSelector* m_apply_u;

};

LineRemover::LineRemover(const double& frequency, const std::
size_t& factor, const std::size_t& order)
: m_frequency(frequency)

```



```

        , m_factor(factor)
        , m_order(order)
        , m_data(0)
    }
}

LineRemover::LineRemover(const UDT& frequency, const UDT&
    factor, const UDT& order)
try : m_frequency(dynamic_cast<const Scalar<double>&>(
    frequency).GetValue())
    , m_factor(dynamic_cast<const Scalar<int>&>(factor).
        GetValue())
    , m_order(dynamic_cast<const Scalar<int>&>(order).GetValue
        ())
    , m_data(0)
{
}
catch (const std::exception& x)
{
    throw std::logic_error(std::string("LineRemover::LineRemover:
        intercepted exception \"" + x.what() + std::string("\")");
}

LineRemover::LineRemover(const LineRemover& lr)
    : m_frequency(lr.m_frequency)
    , m_factor(lr.m_factor)
    , m_order(lr.m_order)
    , m_data(lr.m_data ? lr.m_data->clone() : 0)
{
}

LineRemover::~LineRemover()
{
    delete m_data;
}

LineRemover& LineRemover::operator=(const LineRemover& lr)
{

```

```

if (this != &lr)
{
    m_frequency = lr.m_frequency;
    m_factor = lr.m_factor;
    m_order = lr.m_order;
    delete m_data;
    m_data = lr.m_data ? lr.m_data->clone() : 0;
}
return *this;
}

LineRemover* LineRemover::Clone() const
{
    return new LineRemover(*this);
}

llwd::LdasElement* LineRemover::ConvertTollwd(const CallChain&
Chain, UDT::target_type Target) const
{
    throw General::unimplemented_error(" LineRemover::
ConvertTollwd is unimplemented");
}

void LineRemover::getFilter(UDT*& b) const
try
{
    if (m_data)
    {
        m_data->getFilter(b);
    }
    else
    {
        throw std::logic_error(" LineRemover::getFilter: no filter
estimated yet\n");
    }
}
catch (const std::exception& x)
{

```

```

        throw std::logic_error(std::string(" LineRemover::getFilter:
            intercepted exception \"" + x.what() + std::string("\")");
    }

    template<typename type>
    void LineRemover::refine(const std::valarray<type>& y, const
        std::valarray<type>& u)
    try
    {
        if (!m_data)
        {
            m_data = new Implementation<type>;
        }
        dynamic_cast<Implementation<type>&>(*m_data).refine(y, u,
            m_frequency, m_factor, m_order);
    }
    catch (const std::exception& x)
    {
        throw std::logic_error(std::string(" LineRemover::refine: intercepted
            exception \"" + x.what() + std::string("\")");
    }

    void LineRemover::refine(const UDT& y, const UDT& u)
    try
    {
        if (const std::valarray<float>* p = dynamic_cast<const std::
            valarray<float>*>(&y))
        {
            if (const std::valarray<float>* q = dynamic_cast<const
                std::valarray<float>*>(&u))
            {
                refine(*p, *q);
            }
            else
            {
                throw std::invalid_argument(" LineRemover::refine: input
                    types mismatch");
            }
        }
    }

```

```

    }
  }
  else if (const std::valarray<double>* p = dynamic_cast<const
    std::valarray<double>*>(&y))
  {
    if (const std::valarray<double>* q = dynamic_cast<const
      std::valarray<double>*>(&u))
    {
      refine(*p, *q);
    }
    else
    {
      throw std::invalid_argument("LineRemover::refine: input
        types mismatch");
    }
  }
  else if (const std::valarray<std::complex<float> >* p =
    dynamic_cast<const std::valarray<std::complex<float>
    >*>(&y))
  {
    if (const std::valarray<std::complex<float> >* q =
      dynamic_cast<const std::valarray<std::complex<float>
      >*>(&u))
    {
      refine(*p, *q);
    }
    else
    {
      throw std::invalid_argument("LineRemover::refine: input types
        mismatch");
    }
  }
  else if (const std::valarray<std::complex<double> >* p =
    dynamic_cast<const std::valarray<std::complex<double>
    >*>(&y))
  {
    if (const std::valarray<std::complex<double> >* q =
      dynamic_cast<const std::valarray<std::complex<double>

```

```

        > >*>(&u))
    {
    refine(*p, *q);
    }
    else
    {
    throw std::invalid_argument(" LineRemover::refine: input types
        mismatch");
    }
}
else
{
    throw std::invalid_argument(" LineRemover::refine:
        unsupported types");
}
}
}
catch (const std::exception& x)
{
    throw std::logic_error(std::string(" LineRemover::refine: intercepted
        exception \"" + x.what() + std::string("\""));
}

template<typename type>
void LineRemover::apply(std::valarray<type>& w, const std::
    valarray<type>& u)
try
{
    if (Implementation<type>* p = dynamic_cast<Implementation<
        type>*>(m_data))
    {
        p->apply(w, u);
    }
    else
    {
        throw std::logic_error(" LineRemover::apply: no model yet or
            type mismatch");
    }
}

```

```
}  
catch (const std::exception& x)  
{  
    throw std::logic_error(std::string("LineRemover::apply: intercepted  
        exception \") + x.what() + std::string("\");  
}  
  
void LineRemover::apply(UDT*& w, const UDT& u)  
try  
{  
    if (m_data)  
    {  
        m_data->apply(w, u);  
    }  
    else  
    {  
        throw std::logic_error("LineRemover::apply: must estimate  
            models before applying");  
    }  
}  
catch (const std::exception& x)  
{  
    throw std::logic_error(std::string("LineRemover::apply: intercepted  
        exception \") + x.what() + std::string("\");  
}  
  
LineRemover::Abstraction::Abstraction()  
{  
}  
  
LineRemover::Abstraction::Abstraction(const Abstraction& a)  
{  
}  
  
LineRemover::Abstraction::~Abstraction()  
{
```

```

}

LineRemover::Abstraction& LineRemover::Abstraction::operator=(
    const Abstraction& a)
{
    if (this != &a)
    {
    }
    return *this;
}

template<typename type>
LineRemover::Implementation<type>::Implementation()
: m_model(0)
, m_refine_y(0)
, m_refine_u(0)
, m_apply_y(0)
, m_apply_u(0)
{
}

template<typename type>
LineRemover::Implementation<type>::Implementation(const
    Implementation<type>& lri)
: m_model(lri.m_model ? new OEModel(*lri.m_model) : 0)
, m_refine_y(lri.m_refine_y ? lri.m_refine_y->clone() : 0)
, m_refine_u(lri.m_refine_u ? lri.m_refine_u->clone() : 0)
, m_apply_y(lri.m_apply_y ? lri.m_apply_y->clone() : 0)
, m_apply_u(lri.m_apply_u ? lri.m_apply_u->clone() : 0)
{
}

template<typename type>
LineRemover::Implementation<type>::~Implementation()
{
    delete m_model;
    delete m_refine_y;
    delete m_refine_u;
}

```

```

    delete m_apply_y;
    delete m_apply_u;
}

template<typename type>
LineRemover::Implementation<type>& LineRemover::
    Implementation<type>::operator=(const Implementation<
    type>& Iri)
{
    if (this != &Iri)
    {
        m_model = Iri.m_model ? new OEModel(*Iri.m_model) : 0;
        m_refine_y = Iri.m_refine_y ? Iri.m_refine_y->clone() : 0;
        m_refine_u = Iri.m_refine_u ? Iri.m_refine_u->clone() : 0;
        m_apply_y = Iri.m_apply_y ? Iri.m_apply_y->clone() : 0;
        m_apply_u = Iri.m_apply_u ? Iri.m_apply_u->clone() : 0;
    }
    return *this;
}

template<typename type>
LineRemover::Implementation<type>* LineRemover::Implementation<
    type>::clone() const
{
    return new Implementation<type>(*this);
}

template<typename type>
void LineRemover::Implementation<type>::getFilter(UDT*& b)
    const
{
    if (m_model)
    {
        m_model->getFilterB(b);
    }
    else
    {
        throw std::invalid_argument("LineRemover::Implementation<

```



```

        type>::getFilter: no filter yet");
    }
}

template<typename type> void
LineRemover::Implementation<type>::refine(const std::valarray<type>
    & y,
        const std::valarray<type>& u,
        const double& frequency,
        const std::size_t& factor,
        const std::size_t& order)
{
    // check input sanity

    if (y.size() != u.size())
    {
        throw std::invalid_argument("LineRemover::Implementation<
            type>::refine: model input and output must be the same
            size");
    }

    // construct buffers (will be sized on first call)

    std::valarray<typename complex_traits<type>::complex_type>
        banded_y;
    std::valarray<typename complex_traits<type>::complex_type>
        banded_u;

    if (!m_refine_y) // first call
    {
        m_refine_y = new BandSelector(frequency, factor);
        m_refine_u = new BandSelector(frequency, factor);
        m_apply_y = new BandSelector(frequency, factor);
        m_apply_u = new BandSelector(frequency, factor);
    }

    m_refine_y->apply(banded_y, y);

```

```

m_refine_u->apply(banded_u, u);

if (m_model)
{
    m_model->refine(banded_y, banded_u);
}
else
{
    m_model = new OEModel(banded_y, banded_u, order, 0);
}
}

template<typename T, typename U>
struct aggregator
{
    void operator()(std::valarray<T>&, const std::valarray<U>&)
        const;
};

template<> void aggregator<float, std::complex<float> >::
operator()(std::valarray<float>& out, const std::valarray<std::
complex<float> >& in) const
{
    for (std::size_t i = 0; i < out.size(); ++i)
    {
        out[i] += (in[i].real() * 2);
    }
}

template<> void aggregator<double, std::complex<double> >::
operator()(std::valarray<double>& out, const std::valarray<std::
::complex<double> >& in) const
{
    for (std::size_t i = 0; i < out.size(); ++i)
    {
        out[i] += (in[i].real() * 2);
    }
}

```

```

template<> void aggregator<std::complex<float>, std::complex<
    float> >::operator()(std::valarray<std::complex<float> >& out
    , const std::valarray<std::complex<float> >& in) const
{
    out += in;
}

```

```

template<> void aggregator<std::complex<double>, std::complex<
    double> >::operator()(std::valarray<std::complex<double> >&
    out, const std::valarray<std::complex<double> >& in) const
{
    out += in;
}

```

```

template<typename type> void
    LineRemover::Implementation<type>::apply(
        std::valarray<type>& w,
        const std::valarray<type>& u)
{
    if (!m_apply_u || !m_model || !m_apply_y)
    {
        throw std::logic_error(" LineRemover::Implementation::apply
            Attempted to apply before estimating(refine)");
    }

    std::valarray<typename complex_traits<type>::complex_type>
        banded_u;
    std::valarray<typename complex_traits<type>::complex_type>
        banded_y;
    std::valarray<typename complex_traits<type>::complex_type> y;

    aggregator<type, typename complex_traits<type>::complex_type
        > aggregate;

    m_apply_u->apply(banded_u, u);
}

```

```

    m_model->apply(banded_y, banded_u);
    m_apply_y->ylppa(y, banded_y);
    if (w.size() != y.size()) w.resize(y.size(), type());
    aggregate(w, y);
}

template<typename type>
void LineRemover::Implementation<type>::apply(UDT*& w,
        const UDT& u)
{
    if (const std::valarray<type>* p = dynamic_cast<const std::
        valarray<type>*>(&u))
    {
        if (w == 0)
        {
            w = new Sequence<type>();
        }
        if (std::valarray<type>* q = dynamic_cast<std::valarray<
            type>*>(w))
        {
            apply(*q, *p);
        }
        else
        {
            throw std::invalid_argument(" LineRemover::
                Implementation<type>::apply: input type mismatch")
                ;
        }
    }
    else
    {
        throw std::invalid_argument(" LineRemover::Implementation<
            type>::apply: input type mismatch");
    }
}

#define INSTANTIATE(type) \
\

```

```
template void LineRemover::refine(const std::valarray<type>&,
    const std::valarray<type>&);\
template void LineRemover::apply(std::valarray<type>&, const std::
    valarray<type>&);\
template class LineRemover::Implementation<type>;

INSTANTIATE(float);
INSTANTIATE(double);
INSTANTIATE(std::complex<float> )
INSTANTIATE(std::complex<double> )

#undef INSTANTIATE
}

UDT_CLASS_INSTANTIATION(LineRemover,)
```

Appendix B

Model of a galactic population of continuous wave sources

The validity of the relatively complicated derivation and implementation was tested against a much simpler (but slower and less accurate) Monte-Carlo implementation for an ensemble of detectors and strains.

Listing B.1: Siderially-averaged response

```
function result = sidereal(detector, strain);
% SIDEREAL returns the average (over one sidereal day) of the square of
  the response of a detector to the strain. Note that its value is
  independent of detector longitude.

theta = detector.theta;
psi = detector.psi;

A = zeros(3,3);

A(1,1) = sin(theta).^2.*(cos(psi).^2-sin(psi).^2);
A(1,2) = -2.*sin(theta).*cos(psi).*sin(psi);
A(2,1) = A(1,2);
A(2,2) = sin(psi).^2-cos(psi).^2;

B = zeros(3,3);

B(1,1) = sin(psi).^2-cos(psi).^2;
```

```

B(1,2) = 2.*sin(theta).*cos(psi).*sin(psi);
B(2,1) = B(1,2);
B(2,2) = sin(theta).^2*(cos(psi).^2-sin(psi).^2);

C = zeros(3,3);

C(1,1) = 4.*sin(theta).*cos(psi).*sin(psi);
%C(1,2) = cos(psi).^2 - sin(psi).^2;
C(1,2) = (sin(theta).^2 + 1).*(cos(psi).^2 - sin(psi).^2);
C(2,1) = C(1,2);
C(2,2) = -4.*sin(theta).*cos(psi).*sin(psi);

D = zeros(3,3);

D(1,3) = cos(theta).*sin(theta).*(sin(psi).^2-cos(psi).^2);
D(2,3) = 2.*cos(theta).*cos(psi).*sin(psi);
D(3,1) = D(1,3);
D(3,2) = D(2,3);

E = zeros(3, 3);

E(1,3) = -2.*cos(theta).*sin(psi).*cos(psi);
E(2,3) = sin(theta).*cos(theta).*(sin(psi).^2 - cos(psi).^2);
E(3,1) = E(1,3);
E(3,2) = E(2,3);

F = zeros(3,3);

F(3,3) = cos(theta).^2.*(cos(psi).^2-sin(psi).^2);

A = sum(sum(A.*strain));
B = sum(sum(B.*strain));
C = sum(sum(C.*strain));
D = sum(sum(D.*strain));
E = sum(sum(E.*strain));
F = sum(sum(F.*strain));

```

```
| result = (.75*A*A+.75*B*B+.25*C*C+D*D+E*E+2*F*F+.5*A*B+2  
|         *A*F+2*B*F)*.125; |
```


Bibliography

- [1] <http://www.ldas-sw.ligo.caltech.edu/cgi-bin/cvsweb.cgi/ldas/api/datacondAPI/so/src/>.
- [2] B Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.
- [3] TCL (Tool Command Language). <http://www.tcl.tk>.
- [4] <http://www-unix.mcs.anl.gov/mpi/>.
- [5] J K Blackburn. *The Data Conditioning API's baseline requirements*. Technical Note T990002-00, LIGO, 1999.
<http://www.ligo.caltech.edu/docs/T/T990002-00.pdf>.
- [6] Scott Meyers. *Effective STL*. Addison-Wesley, 2001.
- [7] Andrei Alexandrescu. *Modern C++ Design*. (C++ In-Depth). Addison-Wesley, 2001.
- [8] Boost C++ Library. <http://boost.org>.
- [9] Scott Meyers. *Effective C++*. Addison-Wesley, 2nd edition, 1997.
- [10] Scott Meyers. *More Effective C++*. Addison-Wesley, 1996.
- [11] BLAS (Basic Linear Algebra Subprograms).
<http://www.netlib.org/blas/>.
- [12] CLAPACK (C Linear Algebra Package). <http://www.netlib.org/clapack/>.
- [13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Resusable Object-Oriented Software*. (Professional Computing). Addison-Wesley, 1995.

- [14] B Abbott *et al.* Analysis of first LIGO science data for stochastic gravitational waves. *Phys. Rev. D*, 69(122004), 2004.
- [15] A C Searle, S M Scott, and D E McClelland. Spectral line removal in the LIGO Data Analysis System. *Class. Quant. Grav.*, 20:S721–S730, 2003.
- [16] B Allen, W Hua, and A C Ottewill. Automatic cross-talk removal from multi-channel data. Formal Note P990002-00-E, LIGO, 1999. gr-qc/9909083.
- [17] A M Sintes and B F Schutz. Removing line interference from gravitational wave interferometer data. In S. Kawamura and N. Mio, editors, *Gravitational Wave Detection II*, number 32 in Frontiers Science. Universal Academy Press, 2000. gr-qc/0005071.
- [18] L Ljung. *System identification: theory for the user*. Information and system sciences. Prentice Hall, 2nd edition, 1999.
- [19] <http://www.ldas-dev.ligo.caltech.edu/doc/userAPI/html/actions.html#resample>.
- [20] G Deane. *Honours thesis*. The Australian National University, 200.
- [21] A Lazzarini, R Schofield, and A Vicere. 60 Hz mains correlations for the US power grids. Presentation G020245-00-E, LIGO, 2002.
- [22] <http://www.nasa.gov>.
- [23] Lee Samuel Finn and David F. Chernoff. Observing binary inspiral in gravitational radiation: One interferometer. *Phys. Rev. D*, 47(6):2198–2219, March 1993.
- [24] B Allen. Gravitational wave detector sites. gr-qc/9607075, 1996.
- [25] P Jaranowski and A Krolak. Optimal solution to the inverse problem for the gravitational wave signal of a coalescing compact binary. *Phys. Rev. D*, 49:2198, 1993.
- [26] P Jaranowski, K D Kokkotas, A Krolak, and G Tsegas. On the estimation of parameters of the gravitational-wave signal from a coalescing binary by a network of detectors. *Class. Quantum Grav.*, 13:1279, 1996.

- [27] Lee Samuel Finn. Aperture synthesis for gravitational-wave data analysis: Deterministic sources. *Phys. Rev. D*, 63(102001), April 2001.
- [28] A Pai, S Dhurandhar, and S Bose. Data-analysis strategy for detecting gravitational-wave signals from inspiraling compact binaries with a network of laser-interferometric detectors. *Phys. Rev. D*, 64:042004, 2001.
- [29] D E McClelland, S M Scott, M B Grey, D A Shaddock, B J Slagmolen, A C Searle, D G Blair, L Ju, J Winterflood, F Benabid, M Baker, J Munch, P J Veitch, M W Hamilton, M Ostermeyer, D Mudge, D Ottaway, and C Hollitt. Second generation laser interferometry for gravitational wave detection: ACIGA progress. *Class. Quant. Grav.*, 18:4121–4126, 2001.
- [30] David E. McClelland. (private communication), 2001.
- [31] A Brilliet. (private communication), 2001.
- [32] M Ando. (private communication), 2001.
- [33] N Arnaud *et al*, 2001. gr-qc/0107081.
- [34] B Abbott *et al*. Setting upper limits on the strength of periodic gravitational waves from PSR J1939 + 2134 using the first science data from the GEO 600 and LIGO detectors. *Phys. Rev. D*, 69(082004), April 2004.
- [35] Steve J. Curran and Lorimer Dunc. R. Pulsar statistics — III. Neutron star binaries. *Mon. Not. R. Astron. Soc.*, 276(347–352), 1995.
- [36] B. Abbott *et al*. Analysis of LIGO data for gravitational waves from binary neutron stars. *Phys. Rev. D*, 69(122001), 2004.
- [37] B. Abbott *et al*. First upper limits from LIGO on gravitational wave bursts. *Phys. Rev. D*, 69(102001), 2004.