THE UNIVERSITY
of ADELAIDE

THE APPLICATION OF TIKHONOV REGULARISED INVERSE FILTERING TO DIGITAL COMMUNICATION THROUGH MULTI-CHANNEL ACOUSTIC SYSTEMS

Pierre M. Dumuid

School of Mechanical Engineering
The University of Adelaide
South Australia 5005

Submitted for the degree of Doctor of Philosophy, 26th August 2011.
Accepted subject to minor amendments, 22th November 2011.
Submitted with minor amendments, 3rd February 2012.

Abstract

Communication between underwater vessels such as submarines is difficult to achieve over long distances using radio waves because of their high rate of absorption by water. Using underwater acoustic wave propagation for digital communication has the potential to overcome this limitation. In the last 30 years, there have been numerous papers published on the design of communication systems for shallow underwater acoustic environments. Shallow underwater acoustic environments have been described as extremely difficult media in which to achieve high data rates. The major performance limitations arise from losses due to geometrical spreading and absorption, ambient noise, Doppler spread and reverberation from surface and seafloor reflections (multi-path), with the latter being the primary limitation. The reverberation from multi-path in particular has been found to be very problematic when using the general communication systems that have been developed for radio wave communication systems.

In the early 1990s, the principal means of combating multi-path in the shallow underwater environment was to use non-coherent modulation techniques. Coherent techniques were found to be challenging due to the difficulty of obtaining a phase-lock and also that the environment was subject to fading. Designs have since been presented that addressed both of these problems by using a complex receiver design that involved a joint update of the phase-lock loop and the taps of the decision feedback filter (DFE). In recent years a technique known as time-reversal has been investigated for use in underwater acoustic communication systems. A major benefit of using the time-reversal filter in underwater acoustic communication systems is that it can provide a fast and simple method to provide a receiver design of low complexity.

A technique that can be related to time-reversal and possibly used in underwater acoustics is Tikhonov regularised inverse filtering. The Tikhonov regularised inverse filter is a fast method of obtaining a stable inverse filter design by calculating the filter in the frequency domain using the fast Fourier transform, and was originally developed for use in audio reproduction systems. Previous research has shown that the Tikhonov regularised inverse filter design outperformed time-reversal when using a Dirac impulse

transmission within a simulated underwater environment. This thesis aims to extend the previous work by examining the implementation of Tikhonov regularised inverse filtering with communication signals. In addressing this goal, two topics have been examined: the influence of the sensitivities in the filter designs, and an examination of various design implementations for Tikhonov regularised inverse filtering and similar filtering techniques.

The influence of transducer sensitivities on the Tikhonov regularised inverse filter

During the implementation of the Tikhonov regularised inverse filter it was observed that both the Tikhonov regularised inverse filter and the time-reversal filter were influenced by the sensitivity of the transducers to the acoustic signals, which is determined by the transducer design and the amplifying stages. Unlike single channel systems, setting the sensitivities of the transducers to their maximum value for multi-channel systems does not always maximise the coherence between the input and output of the entire system consisting of the inverse filter, the sensitivities and the electro-acoustic system where the channel is the electro-acoustic transfer function between the transmitter and receiver. The influence the sensitivities have on the performance of the multi-channel Tikhonov regularised inverse filters and the time-reversal filter was examined by performing a mathematical examination of the system. An algorithm was developed that adjusted gains to compensate for the decrease in performance that results from the poor sensitivities. To test the algorithm, a system with an inappropriate set of sensitivities was examined. The performance improvement of the communication system was examined using the generated gains to scale the signal. The algorithm was found to reduce the signal degradation and cross-talk. If the gains were used in the digital domain (after the analog to digital and before the digital to analog converters) then the quality of the signal was improved at the expense of the signal level.

During this examination it was found that the time-reversal filter is equivalent to the Tikhonov regularised inverse filter with infinite regularisation.

Variations of the Tikhonov regularised inverse filter and performance comparisons

In this thesis, various design structures for the implementation of the Tikhonov inverse filter were proposed and implemented in an experimental digital communication system that operated through an acoustic environment in air. It was shown that the Tikhonov inverse filter and related filter design

structures could be classified or implemented according to three different classifications. The Tikhonov inverse filter was implemented according to each of these classifications and then compared against each other, as well as against two other filter designs discussed in the literature: time-reversal filtering, and the two-sided filter developed by Stojanovic [2005]. Due to the number of parameters that could be varied, it was difficult to identify the influence each parameter had on the results independently of the other parameters. A simulation was developed based on a model of the experiment to assist in identifying the influences of each parameter. The parameters examined included the number of transmitter elements, carrier frequency, data rate, and the value of the regularisation parameter.

When the communication system consisted of a signal receiver, the Stojanovic two-sided filter generally outperformed the Tikhonov regularised inverse filter designs when communicating. However, at higher data rates, the Stojanovic two-sided filter required the addition of a regularisation parameter to allow it to continue to operate. However, given an appropriately selected regularisation parameter, the difference between the performance of the Tikhonov filter and the Stojanovic two-sided filter was minimal.

When performing multi-channel communications, the full MIMO implementation of the Tikhonov regularised inverse filter design was shown to have the best performance. For the environment considered, the Tikhonov regularised inverse filter was the only design that was able to eliminate all symbol errors.

Statement of originality

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution to Pierre Dumuid and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis when deposited in the University Library, being made available for loan and photocopying, subject to the provisions of the Copyright Act 1968.

The author acknowledges that copyright of published works contained within this thesis (as listed in Section 1.3) resides with the copyright holder(s) of those works.

I also give permission for the digital version of my thesis to be made available on the web, via the University's digital research repository, the Library catalogue, the Australasian Digital Theses Program (ADTP) and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

Pierre M. Dumuid

Acknowledgements

I would like to acknowledge a number of people without whom this thesis would have never been finished. I firstly wish to acknowledge God who has given me a love and peace I have felt in my life. I am also very thankful to my parents, Bernard and Anthea Dumuid who have encouraged me with my technical interest, and comforted me with emotional, moral, and financial support. I am also very thankful to my sister, Sarah who has been a great sibling, and put up with my moodiness as a house-mate, and helping me to grow into the person I have become.

I wish to also thank my supervisors, Ben Cazzolato, and Anthony Zander, who have had to read though revision after revision of my work, attempting to decipher incomprehensible sentences. Barbara Brougham also provided professional editorial advice in regards to language, completeness and consistency of this work.

Finally, I wish to thank my wife, Kylie and her parents Deb and Barry Foreman who let me use their holiday house to work on my thesis. As well as proof reading my thesis, Kylie has been a great support and encouragement to me as I finished writing up this thesis.

Contents

List of Figures

1 Introduction

The most common mechanism used to achieve wireless communication is radio waves. This can be attributed to the fact that radio waves travel in the earth's atmosphere extremely quickly, and with little absorption. In the underwater environment however, radio waves are absorbed by the ocean at a much greater rate, and are thus only able to be used for very short ranges (of the order of tens of metres). In order to transmit sound over longer distances, a different mechanism needs to be used. The predominant mechanism that has been considered to date is acoustic wave propagation. Acoustic wave propagation has been considered a useful means by which to communicate underwater because the waves travel over large distances with low levels of attenuation. However, using acoustic waves for digital communication systems has a number of short-comings relative to radio waves, particularly slower propagation speed, smaller bandwidth, high level of reverberation, and temporal and spatial variation of the transmission paths [Dunbar, 1972]. Despite these short-comings, increasing interest in underwater communication is evident [Baggeroer, 1984, Catipovic, 1990, Stojanovic, 1996, Kilfoyle and Baggeroer, 2000, Chitre et al., 2008].

The propagation of sound waves in the underwater acoustic environment varies considerably depending on the environmental conditions. The conditions that affect the propagation include the depth, temperature, chemical composition, sea floor composition and also the weather condition. An environment that has been particularly challenging to perform communication in is the shallow water environment. In a shallow water environment sound is subject to multiple reflections from both the surface and the seafloor (often termed reverberation) [Etter, 1996]. When using general communication theory to develop communication systems, these reflections are very problematic. A number of techniques have arisen to overcome the high level of reverberation, and in some cases take advantage of it.

In the literature two groups of researchers have examined the implementation of digital underwater acoustic communication systems for shallow water environments. One group investigated underwater acoustic communication from the basis of general digital communication theory [Baggeroer, 1984, Catipovic, 1990, Stojanovic, 1996, Kilfoyle and Baggeroer, 2000, Chitre

et al., 2008]. The second group have looked at using an acoustic technique known as *time-reversal* and investigated means of integrating this technology with digital communication systems presented [Jackson and Dowling, 1991, Kuperman et al., 1998, Hodgkiss et al., 1999, Kim et al., 2001a]. The time-reversal technique arose from an investigation by Parvulescu [1995] that found that the underwater environment itself could be used to compensate for the reverberation, thus avoiding the need for computationally complex electronic systems that would otherwise be required. Time-reversal has seen much development, and is shown to have many other beneficial properties that shall be discussed further in Section 3.

Another channel compensation technique related to time-reversal is the Tikhonov inverse filter. Tikhonov inverse filtering is a technique that was investigated by Kirkeby et al. [1996a] for use in human listening environments to compensate for room acoustics. Whilst other inverse filter designs exist, the Tikhonov inverse filter provided a means to drastically reduce the computational complexity by performing the calculations in the frequency domain. Cazzolato et al. [2001] compared time-reversal with Tikhonov inverse filtering for use in the underwater environment. The investigations performed by Cazzolato et al. showed that the Tikhonov inverse filter had better spatial and temporal focusing than the time-reversal technique. The work presented by Cazzolato et al. [2001] was an initial investigation of the use of Tikhonov inverse filters for underwater acoustic communication. The purpose of this thesis is to continue the investigation and examine the integration of Tikhonov inverse filters with digital communication systems.

1.1 Aim of this research

The aim of this research is to investigate the implementation and performance of Tikhonov inverse filtering in conjunction with digital communication systems with specific application to shallow water environments. Prior to the commencement of this research, the only previously known work to examine Tikhonov inverse filtering for application in underwater acoustic communication was the work published by Cazzolato et al. [2001]. Cazzolato et al. examined the transmission of a single pulse through a simulated underwater environment. This thesis aims to extend the investigation by examining the implementation of Tikhonov inverse filtering with actual communication signals.

When implementing Tikhonov inverse filtering and time-reversal in communication systems, a number of adjustable parameters exist that include the transducer placement, sensitivity of the transducers, parameters of the inverse filters, design structure of the inverse filter, data rate, and carrier frequency. This research aims to investigate the influence these parameters

have on the system design and its performance.

Two novel contributions have resulted from this research. The first contribution was the finding of a relationship between the transducer sensitivities and performance of the Tikhonov inverse filters, and the second contribution was to provide alternate implementations of the Tikhonov inverse filter along with a comparison of their performance.

As part of this research, an algorithm is also presented that provides a suitable choice of amplifier gains for a given environment. It is also shown that a relationship exists between time-reversal and inverse filtering, whereby time-reversal can be considered equivalent to Tikhonov inverse filtering with a specific choice of the filter parameters.

1.2 Thesis overview

The development of underwater acoustic communication systems requires an understanding of both the propagation of sound in the underwater environment and the theory of digital communication. To assist the reader, the relevant background theory on underwater acoustics and digital communication theory is outlined in Chapter 2. Section 2.1 examines the propagation of sound, and methods used to model underwater acoustics; and Section 2.2 introduces the theory of digital communication, describing how digital data is coded and modulated / demodulated to transmit digital information through an analog channel. Some commonly used receiver structures are also introduced.

Chapter 3 contains a review of the literature that provides a context for the current research. Section 3.1 examines the development of general digital communication for underwater environments, and Section 3.2 examines the development of time-reversal and inverse filtering along with the use of these filters within underwater communication systems.

To investigate the ability to implement Tikhonov inverse filtering in digital communication systems, several experiments were conducted. During these experiments, it was realised that the performance of the Tikhonov inverse filtering was influenced by the magnitude of the gains used at the source and receiver amplifiers. A theoretical analysis of the influence the amplifier gains have on Tikhonov inverse filter designs is described in Chapter 4. A mathematical analysis is provided, along with an algorithm to find the most desirable gains.

The experiments conducted to investigate and validate the implementation of Tikhonov inverse filtering are described in Chapter 5. The purpose of this chapter is to describe the experimental apparatus, computing architecture and computer code and to provide evidence of working code. The details of the theory and the main results obtained from the experiments

and simulations are presented in Chapter 6. A number of design structures for the implementation of the Tikhonov inverse filter are presented, and the performance for each of these structures is compared along with the time-reversal filter, and the Stojanovic [2005] two-sided filter.

Chapter 7 contains the main conclusions that can be drawn from this thesis. Included in this chapter are a number of topics that could be investigated for future research.

1.3 Published material

The published materials resulting from this research are:

Journal papers:

- **Transducer sensitivity compensation using diagonal preconditioning for time reversal and Tikhonov inverse filtering in acoustic systems**
  Pierre M. Dumuid, Ben S. Cazzolato, and Anthony C. Zander,
  Journal of the Acoustical Society of America, Volume 119, Issue 1, pp. 372-381, 2006.

- **A comparison of filter design structures for multi-channel acoustic communication systems**
  Pierre M. Dumuid, Ben S. Cazzolato, and Anthony C. Zander,
  Journal of the Acoustical Society of America, Volume 123, Issue 1, pp. 174-185, 2008.

Conference presentation:

- **Experimental results of time reversal and optimal inverse filtering performed in a one dimensional waveguide**
  Pierre M. Dumuid, Ben S. Cazzolato, and Anthony C. Zander
  146th Meeting of the Acoustical Society of America
  Austin, Texas, USA, November 10th - 14th, 2003.
  Abstract available in Journal of the Acoustical Society of America Volume 114, Issue 4, pp. 2407-2408, October 2003.

2 Background Theory

The development of underwater acoustic communication systems requires an understanding of both the propagation of sound in the underwater environment and the theory of digital communication. The purpose of this chapter is to provide the reader with an introduction to the relevant background theory on acoustic propagation in underwater environments and the theory of digital communications to assist the reader with the relevant literature review (Chapter 3) and the work developed in this research presented in the following chapters. Much of the information in this section has been sourced from a number of textbooks that cover these topics, and are listed at the beginning of each section.

2.1 Underwater acoustics

In order to develop an underwater acoustic communication system, knowledge of the means by which sound travels is required to understand the environment in which the system must operate. This section provides an overview of the propagation of sound in the underwater environment. The material contained in this section is obtained from Etter [1996], Tolstoy and Clay [1987], and Jensen et al. [2000].

2.1.1 Sound absorption

The transmission of sound within the underwater environment is generally limited to frequencies less than 100 kHz for ranges over a kilometre. The reason for this is due to the increasing absorption of sound by the ocean with frequency. The absorption of a harmonic signal with frequency, f, can be understood by considering the energy, $E(f)$, received from a source with a radiation flux, $S(f)$. The energy is given by [Skretting and Leroy 1971, Equation 3]

$$E(f) \quad = \quad S(f) - T(f) - \alpha(f)R \tag{2.1}$$

where $T(f)$ is the geometric spreading loss, $\alpha(f)$ the attenuation coefficient in dB / km and R is the range in kilometres. The attenuation coefficient

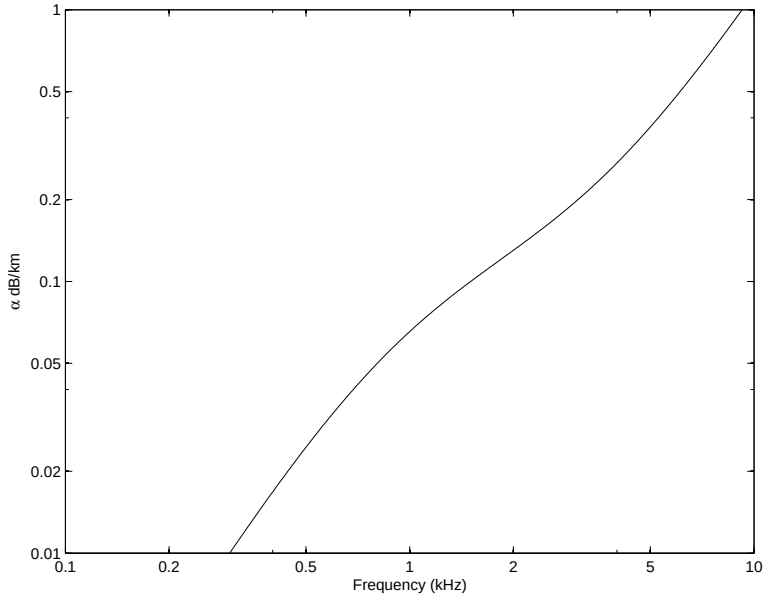


Figure 2.1: Attenuation coefficient of sound in sea-water from the formula given in Equation 2.2.

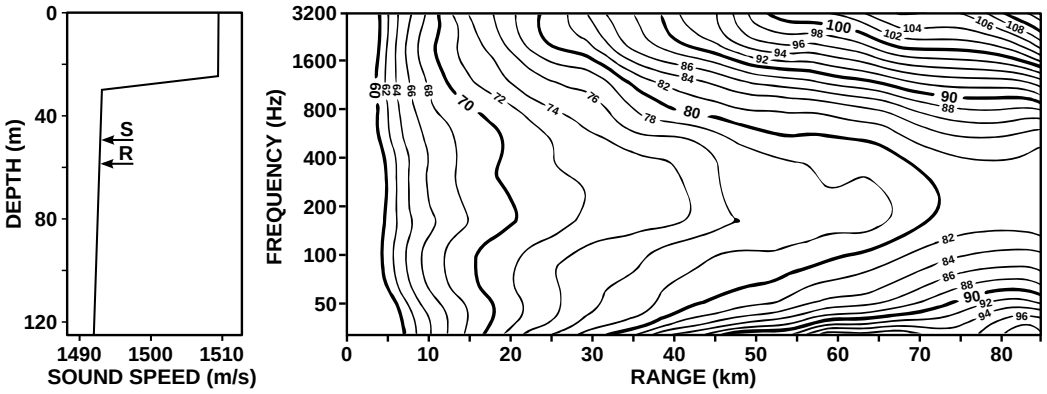
represents the signal energy being absorbed into the medium. A number of authors have proposed formulae that describe $\alpha(f)$. A formula developed by Thorpe [1967], valid for frequencies below 50 kHz is

$$\alpha = 1.094 \left[\frac{0.1 f^2}{1 + f^2} + \frac{40 f^2}{4100 + f^2} \right] \text{ dB/km} \tag{2.2}$$

where f is the frequency in kHz. Figure 2.2 shows the attenuation coefficient using this formula for frequencies between 100 Hz and 10 kHz. From this figure, it is apparent that sound is absorbed at a considerable rate for frequencies above 10 kHz limiting the use of high-frequency transmissions to short ranges.

The geometric spreading loss, $T(f)$, is the attenuation that results from the geometry and is generally derived from the sound speed profile. An example of transmission loss is shown in Figure 2.2b for the shallow water environment with a sound speed profile as per Figure 2.2a. It can be observed that the attenuation across all frequencies is relatively similar at distances up to 20 km. However at extended ranges, the optimal frequency of operation is around 200 Hz and frequencies below 50 Hz and above 500 Hz are highly attenuated.

The geometric spreading loss and absorption by the medium limits the frequency bandwidth over which acoustic communication systems can be used. Sound absorption is the primary limitation for communication at various frequencies and distances. However, even when there is little geometric loss and



(a) Sound Speed Profile

(b) Transmission Loss

Figure 2.2: Experimental measurements of transmission loss (in dB) with respect to range for a shallow environment. The depth of the source and receiver depth are denoted by S and R on sub-figure (a). [Etter 1996, Fig. 5.5,5.6a]

absorption, it is generally difficult to communicate due to the distortion of the transmitted signal that occurs within the ocean.

2.1.2 The wave equation

The propagation of acoustic waves in the underwater environment is governed by the scalar wave equation [Tolstoy and Clay, 1987],

$$\nabla^2 \mathbf{\Phi} = \frac{1}{c^2} \frac{\delta^2 \mathbf{\Phi}}{\delta t^2} \tag{2.3}$$

where ∇^2 is the Laplacian operator, $\mathbf{\Phi}$ the potential or pressure field, c the speed of sound, and t the time. For short intervals of time, the sound speed profile can be considered stationary, and the system considered a time independent system. Under such assumptions, the response of the system to a harmonic source excitation with frequency ω is given by

$$\Phi = \phi e^{-i\omega t} \tag{2.4}$$

where ϕ is the time independent potential function. Substituting Equation 2.4 into Equation 2.3 results in the well known Helmholtz equation

$$\nabla^2 \phi + k^2 \phi = 0 \tag{2.5}$$

where $k = \omega/c$ is the wave number. The Helmholtz equation is also commonly expressed in a cylindrical co-ordinate system as

$$\frac{\delta^2 \phi}{\delta r^2} + \frac{1}{r} \frac{\delta \phi}{\delta r} + \frac{\delta^2 \phi}{\delta z^2} + k^2(z)\phi = 0 \tag{2.6}$$

where z denotes the depth and r the range. The solutions to the wave and Helmholtz equations are used in the modelling methods described in the following section.

2.1.3 Sound propagation modelling

There are five commonly used models based on the Helmholtz equations given in Equations 2.5 and 2.6, being ray theory, normal mode, multi-path expansion, fast-field and parabolic equation techniques [Etter, 1996]. Regardless of the model used to solve the Helmholtz equation, the environmental property that determines the sound propagation is the speed of sound, $c(x, y, z)$. The influence the speed of sound has on acoustic wave propagation is best understood by observing the paths of rays resulting from the ray modelling method. Whilst the ray modelling method is helpful to understanding the propagation of acoustic waves, it is unsuitable for accurate modelling as it is primarily applicable to higher frequency transmissions and / or deep water environments. Thus fast-field modelling, a more practical modelling technique, will also be discussed.

2.1.3.1 Ray theory

Ray theory is developed by taking the solution of the Helmholtz equation to be of the form

$$\phi = A(x, y, z)e^{iP(x,y,z)} \tag{2.7}$$

where $A(x, y, z)$ is an amplitude function, and $P(x, y, z)$ a phase function. After substituting Equation 2.7 into Equation 2.5, and performing a separation of variables according to the real and imaginary parts, the following equalities are obtained:

$$\frac{1}{A}\nabla^2 A - [\nabla P]^2 + k^2 = 0 \tag{2.8}$$

$$2[\nabla A.\nabla P] + A\nabla^2 P = 0. \tag{2.9}$$

Assuming that the variation of the amplitude function is smaller than the wave-number (known as the geometrical acoustic approximation) then it follows that $\frac{1}{A}\nabla^2 A \ll k^2$, and thus

$$[\nabla P]^2 \simeq k^2. \tag{2.10}$$

When $c(x, y, z)$ is known, Equation 2.10 can be used to determine the phase, $P(x, y, z)$, at any location given that $k(x, y, z) = \omega/c(x, y, z)$. The lines of constant phase are known as wave-fronts, and the lines normal to these are called rays.

Although ray tracing can be performed in an environment where the sound speed varies in three dimensions, solving Equation 2.10 in three dimensional environments is computationally expensive and thus ray tracing is generally performed for environments where the sound speed only varies with depth (known as horizontally stratified environments,) or both depth and range. For horizontally stratified environments, rays adhere to Snell's law,

$$\frac{\sin\theta}{c} = a \tag{2.11}$$

where a is a constant, c is the speed of sound, and θ is the angle of the ray with respect to the z-axis. It is of interest to examine the application of Snell's law in two scenarios. The first scenario is that of sound propagating from a medium with a sound speed of c_1 into a medium having a sound speed of c_2. The second scenario is that of sound propagating in a medium where the sound speed varies linearly as a function of depth (i.e. $c(z) = pz$).

When sound propagates from a medium having a sound speed of c_1 into another medium having a sound speed of c_2, Snell's law can be used to arrive at the following relationship:

$$\frac{\sin\theta_1}{c_1} = \frac{\sin\theta_2}{c_2} \tag{2.12}$$

where θ_1 and θ_2 correspond to the angles of the rays in media 1 and 2 as shown in Figure 2.3. The change of angle shown in Figure 2.3 occurs when $c_1 < c_2$. As the wave-fronts travel through the boundary, the wavelength changes according to $\lambda_2 = \frac{c_2}{c_1}\lambda_1$.

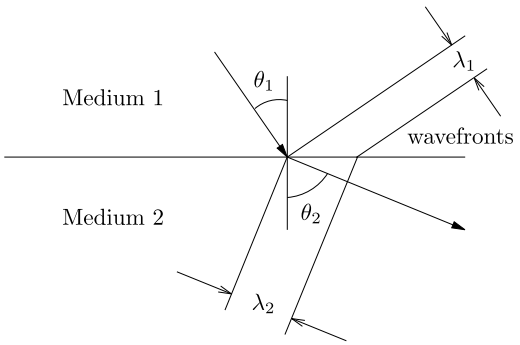


Figure 2.3: The change in angle and wavelength for a ray propagating between two media (Adapted from Etter [1996, Fig. 3.6]).

The second scenario of interest is where the speed of sound is linearly related to the depth (i.e. $c(z) = pz$ where p is a constant). Tolstoy and Clay [1987] show that rays in such an environment follow the trajectory given by

$$x^2 + z^2 = \frac{1}{a^2 p^2} \tag{2.13}$$

where $a = \frac{\sin\theta}{c}$. This equation is observed to be a circular trajectory with a radius of $1/ap$ centred at the depth where $c = 0$. An example of such a ray is shown in Figure 2.4.

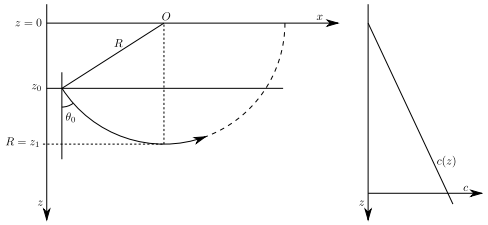


Figure 2.4: The trajectory of a ray in an environment where sound increases linearly as a function of depth. (Adapted from Tolstoy and Clay [1987, Fig.2.15]).

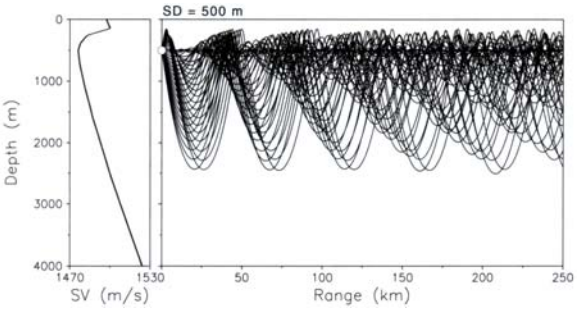
Ray tracing is commonly performed by splitting the sound speed profile of an environment into multiple stacked layers, each layer having a sound speed that is either constant or varies linearly with depth and calculating the trajectory using Equations 2.12 or 2.13. From Figures 2.3 and 2.4 it should be noted that the rays bend towards regions of lower sound speed. Example ray traces are shown in Figure 2.5. These ray traces show the trajectory of rays over a number of departure angles.

Figure 2.5a shows an example of the propagation that occurs for sound emitted in a sound channel, being a local minimum of a sound speed profile. It is evident that most of the energy is trapped within the regions nearest the central axis of the channel, whilst few of the rays reach the deep depths of the ocean, or the surface of the water.

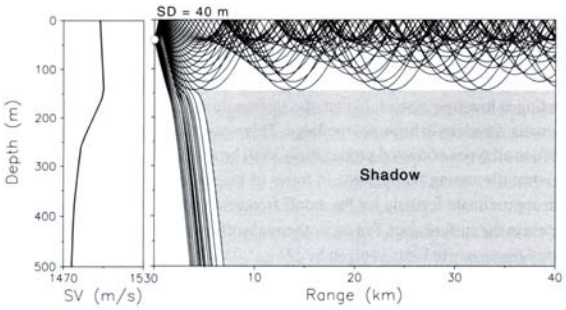
Figure 2.5b shows an example of sound propagation that occurs in an environment known as a surface duct. The surface duct is formed when a local minimum in the sound speed profile is near the surface. As sound propagates through the environment, the rays are refracted towards the surface due to the slower sound speed, and upon hitting the surface, are reflected and then refracted back towards the surface again. The local maximum for this example is at 150 m. The rays that reach the maximum are refracted downwards, causing a region of space that sound originating from a specific source location does not enter. Such a region is known as a shadow zone.

Finally, Figure 2.5c shows an example of sound propagation in a shallow water environment. Sound is reflected at both the surface and the sea floor. Whilst the entire environment is observed to be acoustically excited, the acoustic density of sound is considerably influenced by the sound speed profile.

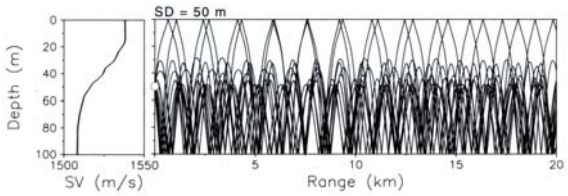
Whilst ray theory is useful for visualising how sound propagates in the underwater environment, this modelling method is less suited to determining



(a) A typical deep water sound speed profile.



(b) A surface duct.



(c) A shallow water environment.

Figure 2.5: Examples of ray tracing for a number of profiles [Jensen et al., 2000, Figs. 1.11-1.14]

underwater acoustic channel responses. In particular, ray tracing is limited by the geometrical acoustic approximation, which requires that the variation of the amplitude function be much smaller than the wave-number [Etter, 1996]. This approximation results in ray tracing being limited to solving higher frequency problems. The regions where the rays become close together (known as caustics) have also been found to be particularly problematic when estimating the amplitude at these locations. Ray tracing is computationally complex as a large number of rays are required to obtain a valid estimate of the sound at certain locations.

2.1.3.2 Normal mode theory

Another means of modelling the underwater environment is by the use of normal modes. The normal mode method of modelling underwater environments was developed by Bucker [1970], and has been described here by relating the solution to the cylindrical Helmholtz equation (Equation 2.6) as described by Jensen et al. [2000] and Etter [1996]. Normal modes are calculated assuming cylindrical symmetry, where the solution to the Helmholtz equation given by [Jensen et al., 2000, Fig. 4.10]

$$\phi(r, z) = \Phi(r)\Psi(z). \tag{2.14}$$

Substituting this equation into the cylindrical Helmholtz equation (Equation 2.6) results in

$$\Psi\frac{d^2\Phi}{dr^2}+\frac{1}{r}\Psi\frac{d\Phi}{dr}+\frac{d^2\Psi}{dz^2}\Phi+k^2\Phi\Psi=0 \tag{2.15}$$

which can be re-arranged using the separation of variables to form

$$\frac{1}{\Psi_m}\left[\frac{d^2\Psi_m}{dz^2}+k^2\Psi_m\right]=\frac{-1}{\Phi_m}\left[\frac{d^2\Phi_m}{dr^2}+\frac{1}{r}\frac{d\Phi_m}{dr}\right]=k_{rm}^2 \tag{2.16}$$

where $\Phi_m(r)$ and $\Psi_m(z)$ are the solution for the range and depth functions for each horizontal propagation constant, k_{rm}^2, $m\in[1,\infty]$. Inserting Equation 2.16 into Equation 2.15 results in [Etter, 1996, Eqns. 4.11 and 4.12]

$$\frac{d^2\Psi_m}{dz^2}+\left(k^2(z)-k_{rm}^2\right)\Psi_m \;=\; 0 \tag{2.17}$$

and

$$\frac{d^2\Phi_m}{dr^2}+\frac{1}{r}\frac{d\Phi_m}{dr}+k_{rm}^2\Phi_m=0. \tag{2.18}$$

Equation 2.17 is known as the depth equation (also called the normal mode equation), and is used to calculate $\Psi_m(z)$, whilst Equation 2.18 is known as the range equation, used to calculate $\Phi(r)$. The range equation is a zero order Bessel differential equation having the solution, $H_0^{(1)}(k_{rm}r)$ the zero-order Hankel function. The depth equation is a *Sturm-Liouville* eigenvalue problem that can be solved for each k_{rm}. Jensen et al. [2000] described a boundary value problem with conditions

$$\Psi(0)=0,\quad \tfrac{d\Psi}{dz}\big|_{z=0} \;=0, \tag{2.19}$$

representative of zero pressure at the surface (typical of an air-water interface), and zero vertical derivative of pressure at the ocean floor (typical of a hard bottom environment). Under these conditions, m represents the number of zeros in the function $\Psi(z)$ over the depth $z=[0,D]$, where D is the depth of the ocean. Jensen et al. [2000] showed that under the boundary conditions given in Equation 2.19, the pressure for a single harmonic source at depth, z_s, is given by [Jensen et al., 2000, Eqn. 5.13]

$$p(r,z) \;=\; \frac{i}{4\rho(z_s)}\sum_{m=1}^{\infty}\Psi_m(z_s)\Psi_m(z)H_0^{(1)}(k_{rm}r) \tag{2.20}$$

where $\rho(z)$ is the density function. It can be observed that the energy contribution for each mode is determined by the product of the magnitudes of the mode shape, $\Psi_m(z)$, at the source depth and receiver depth. Figures 2.6

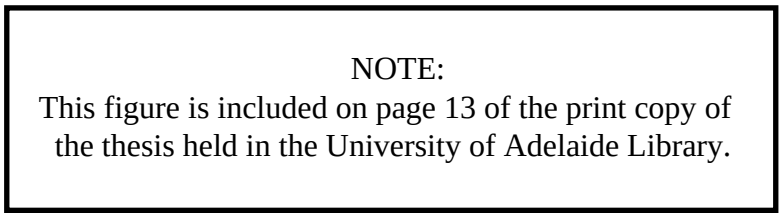


Figure 2.6: Sound speed profile and selected modes of the Pekeris sound-speed profile. The dashed lines are lossy modes that decay with range. [Jensen et al., 2000, Fig. 5.8]

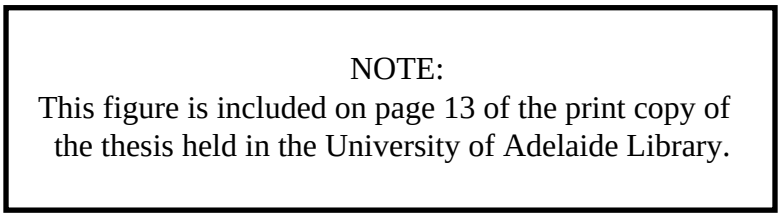


Figure 2.7: Sound speed profile and selected modes of the Munk sound-speed profile for a source frequency of 50 Hz. [Jensen et al., 2000, Fig. 5.10]

and 2.7 show the mode shapes for two different sound speed profiles: a shallow water model, known as a Pekeris profile, and deep water model known as a Munk profile.

Kuperman et al. [1998] noted that the mode functions form a complete set such that

$$\sum_{\text{all modes}} \frac{\Psi_m(z_s)\Psi_m(z)}{\rho(z_s)} = \delta(z - z_s) \tag{2.21}$$

and also satisfy the orthonormal condition,

$$\int_0^\infty \frac{\Psi_m(z)\Psi_m(z)}{\rho(z)} dz = \delta_{nm} \tag{2.22}$$

where δ_{nm} is the Kronecker delta function.

There also exist lossy modes which are modes that decay with range. The Pekeris model shown in Figure 2.6 includes lossy modes that are denoted by the dashed lines. Lossy modes will not be covered in this thesis, and the interested reader is referred to Jensen et al. [2000].

2.1.3.3 Fast-field modelling

Fast-field modelling is a simple fast evaluation of normal modes using the Fast Fourier Transform (FFT). Fast-field modelling involves solving the wave equation by approximating the zero-order Hankel function as

$$H_0^{(1)}(k_{rm}r) \simeq \sqrt{\frac{2}{\pi k_{rm}r}} e^{ik_{rm}r} \tag{2.23}$$

in Equation 2.20. This approximation is valid when $k_{mr}r \gg 1$, and Equation 2.20 then becomes

$$p(r,z) = \frac{i}{4\rho(z_s)} \sum_{m=1}^{\infty} \Psi_m(z_s)\Psi_m(z) \sqrt{\frac{2}{\pi k_{rm}r}} e^{ik_{rm}r} \tag{2.24}$$

and the solution to $p(r,z)$ may then be evaluated by means that utilise the fast Fourier transforms [Etter, 1996].

2.2 Digital communication theory

2.2.1 Introduction

The theory of digital communication is a vast topic, with the first digital communication system being the well known Morse code, developed by Samuel Morse in 1837. Morse code performed communication by the transmission of pulses. Since that time, communication systems have undergone extensive development.

Figure 2.8 shows the general structure of a digital communication system. A digital communication system involves a transmitter and a receiver. The transmitter converts the incoming information into a form suitable to transmit through a physical channel, whilst the receiver converts the signal received from the physical channel and tries to determine the original information that was fed into the transmitter.

The information source consists of a sequence of values. These values are passed into a coder that compresses the data to reduce the amount of digital data transmitted through the channel and / or adds redundant data to improve error resilience. The digital data stream, consisting of a sequence of bits, is then passed into a symbol mapper that maps blocks of bits to a specific symbol in an alphabet, where each symbol denotes a particular waveform that is transmitted. A pulse modulator then converts these symbols into a waveform, having a spectrum that is generally centred around 0 Hz. This waveform is known as a base-band signal. The base-band signal is then passed into a bandpass modulator that shifts the centre frequency of the signal to a frequency more suited for transmission through the channel known as the

Figure 2.8: General overview of a communication system. (Adapted from Sklar [2001, Fig. 1.2])

carrier frequency. The signal at the output of the bandpass modulator is a real-valued signal that is transmitted and received through the physical environment through the means of transducers.

The receiver consists of blocks similar to the transmitter that perform the complementary operation of the transmitter. The receiver converts the signal to a base-band signal, resolves the symbols, maps the symbols into a bit stream, and performs a decoding of the bit sequence to resolve the transmitted data stream.

The following section will examine each of the operations shown in the block diagram in Figure 2.8 in more detail.

2.2.2 Coding

The coder is used in a communication system to modify the incoming data stream prior to transmission. The data stream is generally modified to either compress the data (known as *compression*) by taking advantage of any redundancy in the information stream, or improve error resilience through the addition of redundancy in the data-stream. By introducing error resilience, and enabling the receiver to notify the transmitter of an erroneous data transmission, a communication system can correct for errors. Some systems incorporate coding that increases error resilience to the point that errors can be corrected at the receiver. Such a technique is known as error recovery. By allowing a communication system to operate in a manner which permits errors to occur, the data rate can be increased.

An example of compression is variable-length code encoding. Variable-length encoding involves assigning a different number of bits to each state to be transmitted, based on the probability of each state occurring in the data-stream. Variable-length encoding can be most easily demonstrated by an

example given by Proakis [2001]. If the states are given by a_1, a_2, a_3, a_4 and the probability of each state is $P(a_1) = \frac{1}{2}$, $P(a_2) = \frac{1}{4}$, $P(a_3) = \frac{1}{8}$, $P(a_4) = \frac{1}{8}$, then assigning the bit sequences, $a_1 = 0$, $a_2 = 10$, $a_3 = 110$, $a_4 = 111$ to each of the states results in an average of $(1 * \frac{1}{2} + 2 * \frac{1}{4} + 3 * \frac{1}{8} + 3 * \frac{1}{8}) = 1.75$ bits used for the transmission of each state, which is less than 2, being the number of bits that a binary encoded transmission would use. This reduction of the average number of bits required to transmit data results in a 14.3% increase in the amount of data that can be transmitted in a given period. A method that is often employed to arrive at a variable-length encoding scheme is the Huffman algorithm. Further information concerning the Huffman algorithm and other forms of compression can be found in most communication textbooks, such as Proakis [2001] and Sklar [2001].

An example of coding that allows the detector to determine whether a received data stream contains an error is known as the parity check. The parity check involves grouping the incoming bit stream into blocks of equally sized bits, and adding an extra bit to each block, known as a parity bit. The value of the parity bit is chosen so that there is an even (or odd for odd parity encoding) number of 1's within each block. The receiver examines the blocks and can determine if an error has occurred based on the number of 1's within each block. It should be noted that if an even number of bits are flipped within a block, a false negative occurs (i.e. the decoder considers there to be no error, when in fact there is).

A type of coding that allows for error recovery is linear-block coding. Linear-block codes map blocks of bits in the input stream (a message vector) to a corresponding bit sequence from an alphabet of *code vectors*, where the number of bits in the code vector is greater than the number of bits in each block. If the code vectors are well chosen, the receiver, being able to determine that an error has occurred when a received code vector is not in the alphabet, is able to resolve the received vector to the closest matching vector. If the probability of an error is great, the number of bits in the code vectors would need to be increased to ensure that the vector is resolved appropriately. An example of a Linear Block Code is given is Table 2.1. A single bit error can easily be detected, and resolved to the correct symbol.

Table 2.1: (6,3) Linear Block Code example [Sklar, 2001, Table 6.1]

NOTE:
This table is included on page 17 of the print copy of
the thesis held in the University of Adelaide Library.

Coding has been shown as a means of reducing the data rate through the use of compression, and increasing the error resilience through redundancy. Both techniques can allow for either a smaller bandwidth or a reduced signal-to-noise ratio to transmit the same information. Many other forms of coding techniques exist and the interested reader is referred to standard communication textbooks, such as Proakis [2001] and Sklar [2001], for a more thorough discussion of these topics.

Whilst coding is useful to improve the performance of transmitting data through a channel, the theory of coding has been considered as an area of research outside the scope to which this thesis is devoted. This thesis is aimed at improving the performance of the transmission of the digital information, when it is emitted from the coder and enters into the decoder.

2.2.3 Modulation

2.2.3.1 Base-band and pass-band signals

In order to transmit digital information through a physical channel (such as radio or acoustic wave guides), the digital data is converted into signals that suit the channel so that the signals pass through the environment with minimal signal loss and degradation. As an example, the most appropriate signal for transmission for the environment shown in Figure 2.2 would be a signal that is centred around 200 Hz, particularly for distances greater than 30 km.

The conversion of a digital signal into an analog waveform suitable for transmission in a communication channel is known as modulation. In many communication channels, the frequency at which information is transmitted is generally high, particularly in the case of radio wave communication where the frequencies are of the order of MegaHertz and GigaHertz. To generate such high frequency signals, the signal generation is performed in two stages. The first stage of the modulator generates a complex-valued low frequency

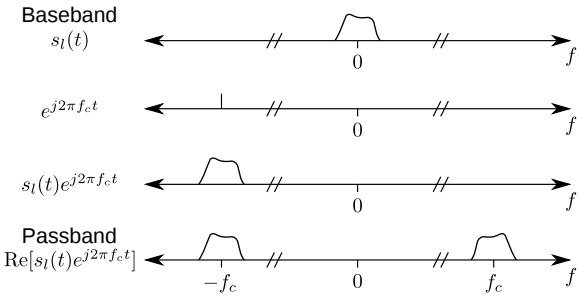


Figure 2.9: Fourier representation demonstrating base-band to pass-band conversion.

signal according to the digital information to be transmitted, known as a base-band signal. The second stage shifts the central frequency of the waveform to the carrier frequency f_c to create a signal for transmission through the environment. This signal is known as a pass-band signal. By splitting the modulation into two separate stages, the low frequency signal generation can use low-cost electrical devices that operate at low sampling rates. The separate stages also enable the transmission frequency to be easily modified independently of the base-band signal generator.

The conversion of a base-band signal, $s_l(t)$, to the pass-band signal, $s(t)$, is given by [Proakis, 2001, Eq. 4.1–14]

$$s(t) = \mathrm{Re}[s_l(t)e^{j2\pi f_c t}]. \tag{2.25}$$

The frequency shift performed in this equation can be confirmed through the rule regarding convolution and Fourier transforms whereby multiplication of two functions in time is equivalent to convolution within the frequency domain. Since $e^{j2\pi f_c t}$ is a delta at $-f_c$ in the frequency domain, the multiplication in Equation 2.25 is a simple shift of $s_l(t)$ in frequency. Given the bandwidth of $s_l(t)$ is much smaller than f_c, the real portion is used as a transmission signal without any loss of information. A graphical representation of this concept is shown in Figure 2.9.

The conversion of a pass-band signal, $r(t)$, to the base-band signal, $r_l(t)$, is given by

$$r_l(t) \quad = \quad \mathrm{LPF}[r(t)e^{-j2\pi f_c t}] \tag{2.26}$$

This conversion involves multiplying the received signal by $e^{-j2\pi f_c t}$ resulting in a frequency shift of f_c, and a low-pass filter (LPF) is used to remove the portion of the spectra centred at $2f_c$. A graphical representation of this concept is shown in Figure 2.10.

The conversion of signals to and from the pass-band signal is generally achieved through a process known as hetero-dyning. A hetero-dyne is the

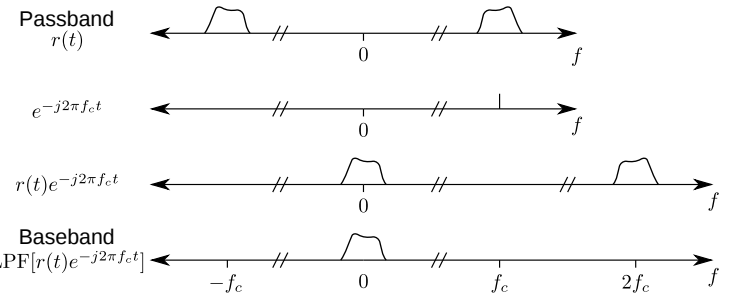
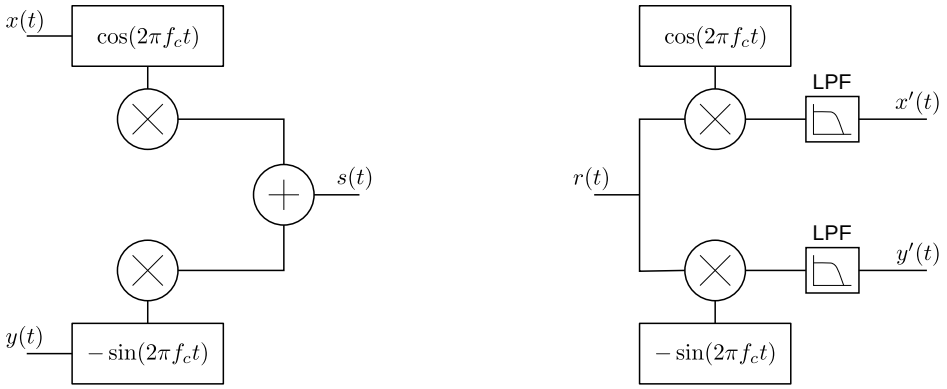


Figure 2.10: Fourier representation demonstrating the pass-band to base-band conversion.



(a) Transmitting hetero-dyne (b) Receiving hetero-dyne

Figure 2.11: Structures for the hetero-dyne operation to (a) convert from base-band to pass-band, and (b) convert from pass-band to base-band

multiplication of a signal by a sinusoidal signal generator. Expressing $s_l(t)$ as $x(t) + y(t)i$ where $x(t)$ and $y(t)$ are the real and imaginary components of $s_l(t)$ respectively, Equation 2.25 can be re-arranged to

$$s(t) = x(t)\cos 2\pi f_c t - y(t)\sin 2\pi f_c t \tag{2.27}$$

which can be used to develop the structure shown in Figure 2.11a. Similarly, the structure to obtain a base-band signal from a pass-band signal is shown in Figure 2.11b.

By expressing $s_l(t)$ in the exponential form $a(t)e^{\theta(t)}$ the transmission signal, $s(t)$, may also be expressed as

$$s(t) = a(t)\cos\left[2\pi f_c t + \theta(t)\right]. \tag{2.28}$$

Assuming that the bandwidth of the signal is much smaller than the carrier frequency, f_c, then $a(t)$ can be seen as the amplitude function, and $\theta(t)$ the phase function.

2.2.3.2 The base-band channel response

Most communication channels, including the underwater acoustic environment, can be considered as LTI (Linear Time Invariant) for short time intervals. Such channels can be modelled as an FIR (Finite Impulse Response) filter with the addition of noise to the system, and the relationship between the transmitted signal, $s(t)$, and the received signal, $r(t)$, given by

$$r(t) = s(t) * h(t) \tag{2.29}$$

where $h(t)$ is the channel response. In the frequency domain, this equates to

$$R(f) = S(f)H(f) \tag{2.30}$$

A similar relation can be obtained between the low-pass transmitted and received signal. Proakis [2001] showed that an equivalent low-pass response is given by

$$R_l(f) = H_l(f)S_l(f) \tag{2.31}$$

where

$$H_l(f) = \begin{cases} H(f + f_c) & f \geq -f_c \\ 0 & f < -f_c \end{cases} \tag{2.32}$$

which is known as the base-band frequency response of the system, having a corresponding time domain response, $h_l(t)$, which is complex valued. It should be noted however that the signals being transmitted are generally band-limited to $B \ll f_c$ and thus the channel response is only required to be known for the spectrum over which it is transmitted,

$$H_l(f) = \begin{cases} H(f + f_c) & |f| \leq B \\ 0 & |f| > B \end{cases} \tag{2.33}$$

2.2.4 Communication signals

The two most common waveforms used to transmit information are frequency shift keying (FSK) and phase / amplitude modulation.

FSK modulation involves transmitting a sinusoidal waveform having a different frequency according to the data to be transmitted. The base-band waveforms for an equally spaced (in terms of frequency) set of waveforms commonly used to implement FSK are given by

$$s_i(t) = e^{j\omega_i t} \quad \begin{matrix} 0 \leq t \leq T \\ i = 1, \ldots, M \end{matrix} \tag{2.34}$$

where $\omega_i = \frac{1}{2}\Delta\omega I_n$, and $I_n = \pm 1, \pm 3, \ldots, \pm(M-1)$.

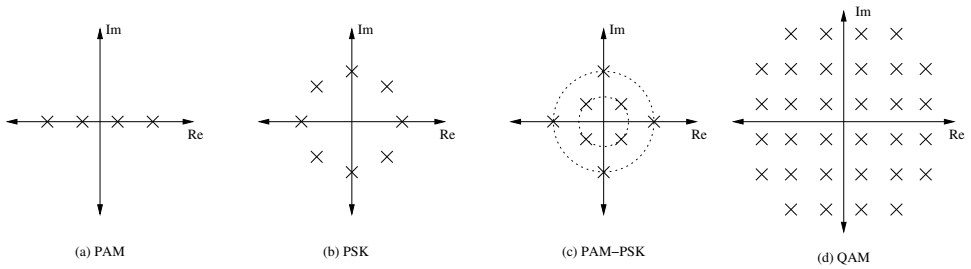


Figure 2.12: Signal space diagrams for a number of modulation techniques

Phase / amplitude modulation encompass a number of modulation waveforms that include Pulse Amplitude Modulation (PAM), Phase Shift Keying (PSK), the combination of both of these (PAM-PSK), and Quadrature Amplitude Modulation (QAM). For each of these modulation techniques, the base-band waveforms are defined as

$$s_i(t) = A_i e^{j\omega_i} g(t) \quad \begin{matrix} 0 \le t \le T \\ i = 1, \ldots, M \end{matrix} \tag{2.35}$$

where $g(t)$ is a spectral shaping filter that will be discussed in Section 6. For PAM, the phase is fixed and the amplitude for each waveform is $A_i = \frac{1}{2}\Delta A I_n$ where $I_n = \pm 1, \pm 3, \ldots, \pm(M-1)$, and M is an even integer. For PSK, the amplitude is fixed and the phase is given by $\omega_i = \frac{1}{2}\Delta\omega I_n$, where $I_n = \pm 1, \pm 3, \ldots, \pm(M-1)$, and M is an even integer. PAM-PSK encompasses modulation techniques where the phase and amplitude are varied in fixed steps. A number of other signal-space constellations are also used that are called QAM. Figure 2.12 shows some examples of each of these modulation techniques, and Figure 2.13 shows example base-band and pass-band waveforms for PAM, PSK, and PAM-PSK when the spectral shaping filter, $g(t)$, is given by

$$g(t) = \begin{cases} 0 & t < -\frac{T}{2} \\ 1 & -\frac{T}{2} \le t \le \frac{T}{2} \\ 0 & \frac{T}{2} \le t \end{cases} \tag{2.36}$$

To convert a sequence of symbols, $I(n)$, transmitted at a symbol rate of $1/T$, to a base-band signal, the following formula can be used

$$s_l(t) \quad = \quad s_1(t) * g(t) \tag{2.37}$$

where

$$s_1(t) = \sum_n I(n) * \delta(t - n * T - T/2). \tag{2.38}$$

The waveforms $s_l(t)$ and $g(t)$ used for the PAM-PSK example shown in Figure 2.13 are presented in Figure 2.14.

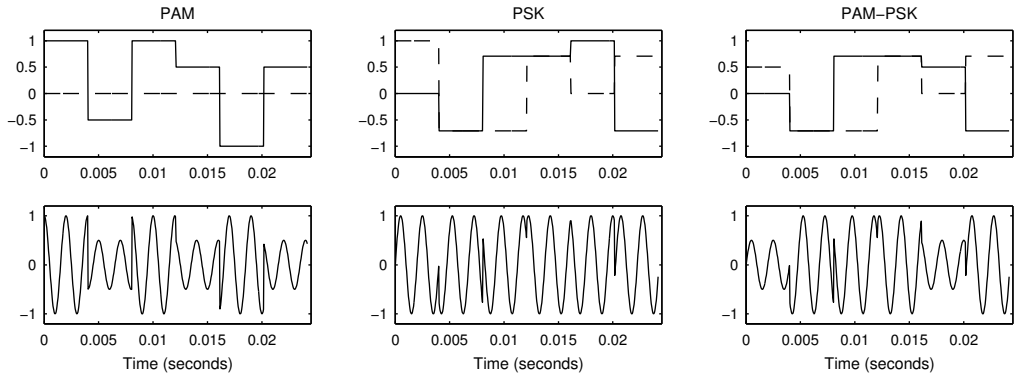
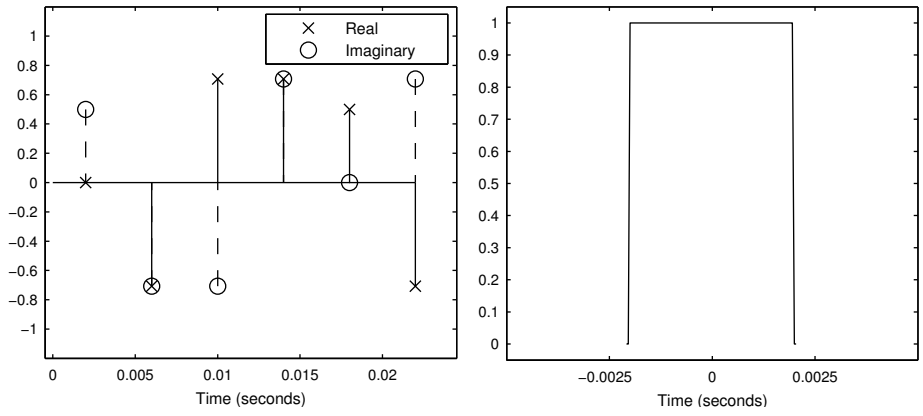


Figure 2.13: Examples of base-band and pass-band signals for PAM, PSK, and PAM-PSK. The top plot shows the base-band with the solid and dashed lines representing the real and imaginary components respectively, whilst the lower plots show the corresponding pass-band signals, for $f_c = 500$ Hz.



(a) (b)

Figure 2.14: Two signals, $s_1(t)$ and $g(t)$, that may be used to generate the base-band signal.

In Figure 2.13 it can be seen that many discontinuities exist. These discontinuities result in excess bandwidth use. To reduce the bandwidth used, one can choose an alternate function for $g(t)$. Assuming that the decoder samples the signal at $t = nT+T/2$, then to ensure that the waveform for each symbol does not interfere with the other symbols at their sampling instances, the function must be constrained so that

$$g(nT) = \begin{cases} 1 & n = 0 \\ 0 & n \neq 0 \end{cases} \tag{2.39}$$

The functions that satisfy this condition are known as Nyquist filters. The Nyquist filter that provides the smallest bandwidth ($W = \frac{1}{T}$) is given by

$$g(t) = \frac{\sin(\pi t/T)}{\pi t/T} = \text{sinc}(\frac{\pi t}{T}) \tag{2.40}$$

and is shown in Figure 2.15. The impulse response for this filter is difficult to implement as it takes some time to decay in both the forward and negative time. Such a long decay requires that the symbols to be transmitted are known far in advance of transmission. The long decay time also increases the inter-symbol interference that results if the signal is sampled at an incorrect time. To avoid such a long decay, a raised-co-sine filter was developed, that is given by

$$g(t) = \frac{\sin(\pi t/T)}{\pi t/T} \frac{\cos(\pi \beta t/T)}{1 - 4\beta^2 t^2/T^2} \tag{2.41}$$

where β is known as the roll-off factor and is set within the range $0 \leq \beta \leq 1$. This filter is a Nyquist filter, and has a frequency response

$$G(f) = \begin{cases} T & 0 \leq |f| \leq \frac{1-\beta}{2T} \\ \frac{T}{2}\left\{1 + \cos\left[\frac{\pi T}{\beta}\left(|f| - \frac{1-\beta}{2T}\right)\right]\right\} & \frac{1-\beta}{2T} \leq |f| \leq \frac{1+\beta}{2T} \\ 0 & \frac{1+\beta}{2T} < |f| \end{cases}. \tag{2.42}$$

The roll-off factor increases the rate of decay at the expense of using more bandwidth, and the corresponding bandwidth is $(1+\beta)\frac{1}{T}$. When $\beta = 0$ the raised co-sine filter is the optimal Nyquist filter given by Equation 2.40. Figure 2.16 shows the impulse and frequency response for various values of β.

Figure 2.17 shows the base-band and corresponding pass-band signals when a raised co-sine spectral shaping filter is employed with $\beta = 0.2$, and the filter is truncated at $\pm 6T$. It may be observed that discontinuities no longer exist for both the base-band and pass-band signals. At the sampling instances (given by the crosses and circles) the signal measured is the same as that given in Figure 2.13.

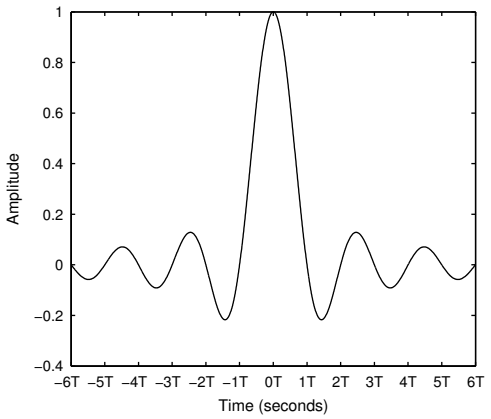


Figure 2.15: Impulse response of the ideal spectral shaping filter

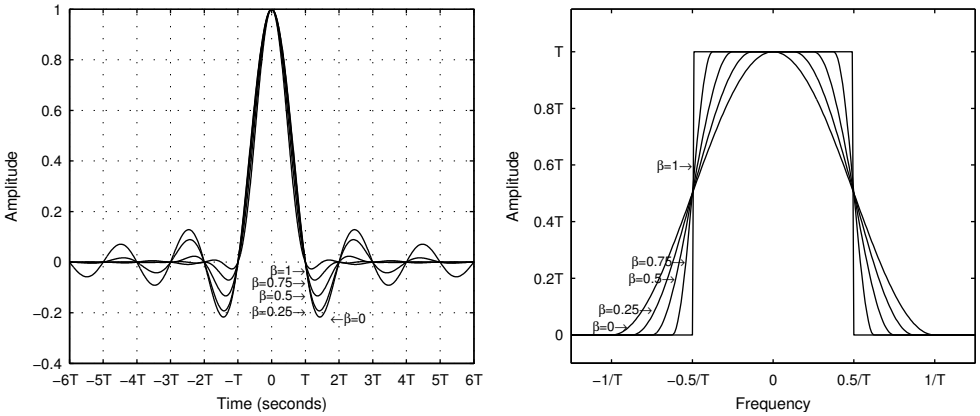


Figure 2.16: Impulse and frequency response of the raised co-sine spectral shaping filter

2.2.5 Receiver structures

In the previous sections it has been assumed that the receiver signal was the same as the transmitted signal, such that there was no delay, or degradation of the signal. Such channels rarely exist in practise, and extra operations are required to compensate for the channel response in order to obtain the base-band signal. The following section will examine some of these operations.

2.2.5.1 Carrier phase recovery

Most channels contain a considerable delay. A delay can be modelled as

$$h(t) = \delta(t - \tau) * h'(t) \tag{2.43}$$

where $*$ is the convolution operator, τ the delay and $h'(t)$ the impulse response of the channel without the delay. In the frequency domain, this

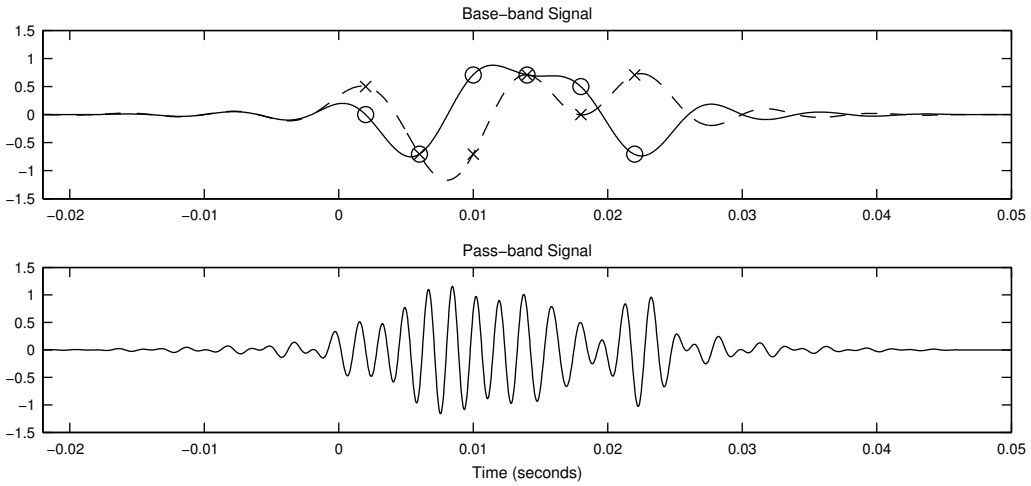


Figure 2.17: Example of base-band and pass-band signals as per the PAM-PSK example shown in Figure 2.13 using a raised co-sine spectral shaping filter with $\beta = 0.2$ truncated at $\pm 6T$. The crosses and circles indicate the sampling instances.

relationship can be represented as

$$H(f) = e^{-j2\pi f\tau} H'(f). \tag{2.44}$$

Substituting this into Equation 2.33 the low-pass channel response is

$$H_l(f) = \begin{cases} e^{-j2\pi(f+f_c)\tau} H'(f+f_c) & |f| \leq B \\ 0 & |f| > B \end{cases}. \tag{2.45}$$

Now

$$e^{-j2\pi(f+f_c)\tau} = e^{-j2\pi f\tau} e^{-j2\pi f_c\tau} \tag{2.46}$$

and thus

$$h_l(t) = e^{-j2\pi f_c\tau} \delta(t-\tau) * h'_l(t) \tag{2.47}$$

$$= e^{-j2\pi f_c\tau} h'_l(t-\tau) \tag{2.48}$$

which can be observed as a delayed version of the base-band impulse response with the addition of a phase shift. In most communication systems, the phase and delay are compensated independently so that the channel is modelled as

$$h_l(t) = e^{-j2\pi f_c\phi} h'(t-\tau) \tag{2.49}$$

where ϕ represents the phase, and τ represents the delay estimate.

NOTE:
This figure is included on page 26 of the print copy of the thesis held in the University of Adelaide Library.

Figure 2.18: Schematic of a QAM receiver [Proakis, 2001, Fig. 6.1-4]

Assuming that $h'(t-\tau) \simeq \delta(t)$, there are a number of methods used to recover the carrier phase. One method involves multiplexing a *pilot signal* within the transmission stream. The receiver can then use a phase-locked loop (PLL) on this pilot signal to determine and track the phase of the carrier. Another technique is to use a carrier phase estimator that uses the received signal and determines the optimal phase that closely matches the expected response.

Figure 2.18 shows an example receiver structure that employs carrier phase recovery. The signal initially passes through an Automatic Gain Control (AGC) which is used to compensate for variations in the amplitude of the received signal. The phase recovery block is then used to phase shift the hetero-dyne so that the correct base-band signal is attained.

2.2.5.2 The matched filter and the noise whitening filter

In order to detect the symbols that are transmitted, a filter is used to maximise the signal at each sampling instant. For a signal transmitted with a spectral shaping filter, $g(t)$, in an environment that consists of additive Gaussian noise, the filter that maximises the signal is known as the matched filter and is given by [Haykin, 2001, Eq. 4.16]

$$f(t) = g^*(-t). \tag{2.50}$$

However when implementing a filter, it cannot be defined for negative time. To create a filter that can be implemented, a delay can be included and only a portion of the duration is used to form the matched filter. Such a filter is given by

$$f(t) = \begin{cases} g^*(\tau - t) & 0 \leq t \leq \tau \\ 0 & \text{elsewhere} \end{cases} \tag{2.51}$$

where τ is known as the duration of this filter and is generally chosen such that the impulse response is negligible after the time τ, and the sampling instant at which the maximum signal to noise ratio is achieved is $t = \tau$.

Whilst the matched filter maximises the SNR at the sampling instance, the noise from the channel is also filtered. If the noise at the input to the matched filter is Gaussian, the noise at the output of the matched filter will have a spectrum similar to that of the matched filter, $F(f)$. This filtering can possibly result in errors when detecting the symbol. To retain the maximal SNR at the sampling instant and flatten the spectrum, a filter known as a noise-whitening filter can be used. The noise-whitening filter is derived by observing that the symbol and the matched filter combined is the auto-correlation function,

$$X(z) = F(z)F^*(z^{-1}) \tag{2.52}$$

A property of the auto-correlation function is that if ρ is a root of $X(z)$ then $1/\rho^*$ is also a root, thus if there is a root of $X(z)$ that is within the unit circle, a corresponding root exists outside the unit circle. If $F(z)$ is chosen such that all the roots are outside the unit circle, then the filter given by $1/F^*(z^{-1})$ is stable and known as the noise-whitening filter, and when used in conjunction with the matched filter, results in the noise-whitened matched filter.

2.2.5.3 Channel compensation

In the previous sections the channel has been modelled as a delay with additive Gaussian noise. Such a model is inadequate for correctly modelling the underwater environment, and the model typically used is a LTI system that is stationary for short periods of time. When transmitting a sequence of waveforms for each symbol in a LTI channel, the channel response distorts the signal such that each waveform overlaps the subsequent transmitted waveforms. Such over-lapping is known as ISI (Inter-Symbol Interference). To compensate for the more complex LTI model, more complex systems may be required to both track the phase of the signal, and also to undo the ISI.

If the channel response were time-invariant, then the response of the channel could be measured, and a fixed filter used to compensate for the channel response. However, when the channel is time-variant, an adaptive form of channel compensation is required to continually track and compensate the variations of the channel response.

2.2.5.4 Channel compensation for time-invariant channels

The linear equalisers One means by which ISI can be compensated is through the use of linear equalisers. A number of linear equalisers exist. In this thesis, the design of an equaliser by the *mean-square-error (MSE)*

criterion shall be discussed. The MSE equaliser is designed on the basis of minimising the mean square error of

$$\epsilon = I_k - \hat{I}_k \tag{2.53}$$

where I_k is the symbol transmitted, and $\hat{I}_k$ is the estimate at the output of the equaliser. The equaliser filter for the combination of the channel and a matched filter, designed from the MSE criterion, is given by [Proakis, 2001, Eq. 10.2–33]

$$C(z) = \frac{1}{X(z) + N_0} \tag{2.54}$$

where N_0 is the spectral density of the noise from the channel, and $X(z)$ is the auto-correlation function of the channel response. Depending on the channel, such an inverse can require an infinite number of taps. In practise only limited number of taps are available, and thus a finite-length equaliser is used. A method of obtaining suitable coefficients for a finite number of taps is given by [Proakis, 2001, Sec. 10.2.2].

2.2.5.5 Adaptive equalisers – Channel compensation for time-variant channels

Time-variant channels can generally be modelled as a stationary channel for short periods of time, where "short periods of time" refers to a duration of at least tens of symbol intervals. To compensate for such a time-invariant channel, a filter known as an *adaptive equaliser* is generally used.

The general structure of an adaptive equaliser is shown in Figure 2.19. The input signal to the adaptive filter structure is a sampled sequence of values from the low-pass signal sampled at the symbol rate or a fraction of the symbol rate. The signal is then filtered by a feed-forward filter. If the sample rate of the input signal was a fractional sampling rate of the symbols, the signal is re-sampled at the symbol rate and combined with a signal from the feed-back filter that operates on the previously detected symbols. The mixed signal is then used on a decision device that determines the most likely symbols that were transmitted. The error between the detected symbol and the measured symbol is used to adjust the filter taps within both the feed-forward and feed-back path. When the channel is unknown, a training sequence is often used instead of the detected symbols to provide an error signal to initially adjust the filters. It should be noted that some adaptive filter configurations do not include the feed-back filter.

Zero forcing algorithm A filter designed for use in a static channel having negligible noise is the zero-forcing equaliser [Proakis, 2001]. The zero-forcing

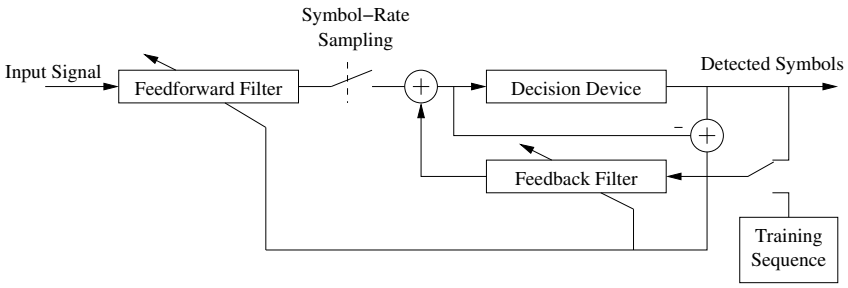


Figure 2.19: Adaptive filtering structure

filter has a corresponding adaptive implementation that can be used in systems involving dynamically varying channels. The input sampled signal, v_k, is passed through an FIR filter to obtained a filtered signal

$$\hat{I}_k = \sum_{j=-K}^{K} c_j^{(k)} v_{k-j} \tag{2.55}$$

where $c_j, j \in [-K, K]$ are the co-efficients of the feed-forward filter. The filtered signal is then used with a detector and results in a detected sequence, $\tilde{I}_k$. The zero-forcing algorithm updates the co-efficients such that the cross correlation between the error sequence, $\varepsilon_k = \tilde{I}_k - \hat{I}_k$, and the detected information sequence, $\left\{\tilde{I}_k\right\}$, over the range of tap filters is zero. The taps of the filter are updated according to the formula [Proakis, 2001, Eq. 11.1–5]

$$c_j^{(k+1)} = c_j^{(k)} + \triangle \varepsilon_k \tilde{I}_{k-j}^{*}. \tag{2.56}$$

Since the future symbols are unknown, it follows that $\tilde{I_{k-j}}$ is only known provided that $j \geq 0$. Thus the adaptive filter co-efficients, c_j, are only defined for $j \geq 0$ and the filter can only operate over the current and past sampled signal values.

The decision-feedback least-mean square (LMS) algorithm A filter designed for use in a static channel having noise is the mean square error (MSE) equaliser [Proakis, 2001]. The static MSE filter has a corresponding adaptive filter implementation known as the least mean square (LMS) adaptive equaliser. The structure of the decision-feedback LMS algorithm adaptive filter is shown in Figure 2.20. The filtered signal, $\hat{I}_k$, used to estimate the transmitted symbols is the result of filtering from both the sampled signal, v_k, and the past detected symbols, $\tilde{I}_{k-j}$. The value for the sampled signal, $\hat{I}_k$, at the sampling instance, k, is given by [Proakis, 2001, Eq. 10.3–1]

$$\hat{I}_k = \sum_{j=-K_1}^{0} c_j^{(k)} v_{k-j} + \sum_{j=1}^{K_2} c_j^{(k)} \tilde{I}_{k-j} \tag{2.57}$$

NOTE:
This figure is included on page 30 of the print copy of
the thesis held in the University of Adelaide Library.

Figure 2.20: Schematic for the decision-feedback MSE equaliser [Proakis, 2001, Fig 11.2–1]

where $c_j, j \in [-K_1, 0]$ are the co-efficients of the feed-forward filter, and $c_j, j \in [1, K_2]$ the co-efficients of the feedback filter. The LMS algorithm uses the steepest descent algorithm to minimise the MSE, resulting in a co-efficient update equation [Proakis, 2001, Eq. 11.1-11]

$$\tilde{\mathbf{C}}_{k+1} = \tilde{\mathbf{C}}_k + \triangle \varepsilon_k \mathbf{V}_k^* \tag{2.58}$$

where

$$\begin{aligned} \tilde{\mathbf{C}_k} &= [\ c_{-K_1}^{(k)} \quad c_{-K_1+1}^{(k)} \quad \cdots \quad c_0^{(k)} \quad c_1^{(k)} \quad \cdots \quad c_{K_2}^{(k)}\ ], \\ \mathbf{V}_k &= [\ v_{k+K_1} \quad v_{k+K_1-1} \quad \cdots \quad v_k \quad I_{k-1} \quad \cdots \quad I_{k-K_2}\ ], \end{aligned}$$

$\triangle$ is the parameter that controls the speed of adaption, and $\varepsilon_k = I_k - \hat{I}_k$ is the error signal between the filtered and detected signal.

Whilst the MSE adaptive filter is a commonly implemented adaptive equaliser, other improved algorithms have been proposed that help speed up the adaption of the equaliser, however they will not be included here.

The recursive least-square (RLS) algorithm The steepest-descent LMS algorithm has slow convergence. An algorithm that has faster convergence is the recursive least-square (RLS) algorithm. The RLS algorithm achieves faster convergence by using an algorithm that minimises a time-average of the error rather than the instantaneous error. The algorithm used to compute the RLS is as follows (taken from Proakis [2001]):

1. Compute the output of the feed-forward and feedback filter:

$$\hat{I}(k) = \mathbf{V}'(k)\mathbf{C}(k-1) \tag{2.59}$$

where $\mathbf{C}_k = [\ c_{-K_1}^{(k)} \quad c_{-K_1+1}^{(k)} \quad \cdots \quad c_0^{(k)} \quad c_1^{(k)} \quad \cdots \quad c_{K_2}^{(k)}\ ]$ are taps of the feed-forward and feedback filter, and

$$\mathbf{V}_k = [\ v_{k+K_1} \quad v_{k+K_1-1} \quad \cdots \quad v_k \quad I_{k-1} \quad \cdots \quad I_{k-K_2}\ ]$$

are input values and detected symbol values.

2. Perform the detection:

$$\tilde{I}(k) \quad = \quad \begin{cases} I(k) & \text{training} \\ E\{\hat{I}(k)\} & \text{non} - \text{training} \end{cases} \tag{2.60}$$

where $E\{\}$ denotes a function to find the closest symbol using the closest Euclidean distance method.

3. Compute the error:

$$e(k) \quad = \quad \tilde{I}(k) - \hat{I}(k) \tag{2.61}$$

4. Compute the *Kalman gain vector*, and update the matrix, $\mathbf{P}(k)$:

$$\mathbf{K}(k) \quad = \quad \frac{\mathbf{P}(k-1)\mathbf{V}^*(k)}{w + \mathbf{V}'(k)\mathbf{P}(k-1)\mathbf{V}^*(k)} \tag{2.62}$$

$$\mathbf{P}(k) \quad = \quad \frac{1}{w}\left[\mathbf{P}(k-1) - \mathbf{K}(k)\mathbf{Y}'(k)\mathbf{P}(k-1)\right] \tag{2.63}$$

where $\mathbf{K}(k)$ is a gain vector used when updating the filter taps, w is a weighting factor and $\mathbf{P}(k)$ is a matrix that is introduced in the derivation of the algorithm, and reduces the number of inversions required by being updateable from the previous state of the matrix. The weighting factor, w, is an exponential weighting of the past error values and is required to be in the range $0 < w < 1$.

5. Update the filter co-efficients:

$$\mathbf{C}(k) \quad = \quad \mathbf{C}(k-1) + \mathbf{K}(k)e(k) \tag{2.64}$$

The algorithm can be initialised with the matrix, $\mathbf{P}(0) = \mathbf{I}$.

Concluding Statement

The background theory presented in this chapter provides a platform for the concepts presented in the literature review (Chapter 3), and for further development of the theory in Chapters 4 to 7. The content presented in this chapter is indirectly related to the work performed in this thesis in that it provides relevant background theory required to better understand literature related to and the works conducted for this thesis.

3 Literature Review

This chapter provides a review of the literature pertinent to the work undertaken in the subsequent chapters of this thesis on the design and development of underwater acoustic communication systems using Tikhonov regularised inverse filtering. The literature on the development of underwater communication systems has been examined in the literature from two angles, that using general digital communication theory, and that which has used channel compensation techniques such as time-reversal. In Section 3.1, the literature concerning the development of underwater acoustic communications using general digital communication theory is described. The literature concerning the development of channel compensation techniques is presented in Section 3.2, and Section 3.3 describes the literature concerned with the application of channel compensation to acoustic communication systems.

3.1 Digital underwater acoustic communication

Over the years, a number of papers [Baggeroer, 1984, Catipovic, 1990, Stojanovic, 1996, Kilfoyle and Baggeroer, 2000, Chitre et al., 2008] have summarised the development of underwater acoustic communication, a branch of telemetry dealing with the transmission of data through water. Each review has provided an overview of the state of development at that point in time, often accompanied by a theoretical description of some particular problem.

The overview by Kilfoyle and Baggeroer [2000] in particular provides a good historical perspective of the development in this field. Kilfoyle and Baggeroer [2000] point out that prior to technological developments in the 1970s there were few published reports of underwater telemetry, and that those reports that were provided essentially described an underwater loudspeaker because of the inability to mitigate multi-channel sound dispersion. It was not until the late 1970s that developments in digital signal processing resulted in telemetry systems that could perform error correction, and compensate for channel reverberation in underwater communication systems.

Baggeroer [1984] discussed developments in underwater acoustic commu-

nication up until that time. The implementation of communication systems in underwater environments was found to have many problems, such as path losses, ambient noise, reverberation and Doppler effects.

Of these problems, reverberation was found to be the most challenging to overcome. Frequency shift keying (FSK) and *single level amplitude modulation* were the primary means used to overcome reverberation in the early communication systems. FSK was reliable but suffered from high power requirements and was also inefficient at utilising the available bandwidth. In particular, to avoid the influence of multi-path, guard times (time of silence between transmissions) had to be used. Amplitude modulation was found to be suitable in a few environments when the channel was 'clean' and reverberation was low.

Advances in microprocessors to allow the use of fast Fourier transforms (FFT) to perform processing were seen as a means of providing improved performance in the future. In some instances, transmission via a spread spectrum was used to overcome fading. To avoid multi-path dispersal of sound, a suggestion was made that very narrow beams be used to transmit or receive along a single path. However such beam-forming requires a large array, high transmitter power, and exhibits pointing error when the transmitter array and receiver array beams do not match.

The highest data rates reported by Baggeroer [1984] were from systems using FSK and/or parametric arrays to transmit along very narrow beams. Mention was made of other developments in the classified literature, but no details were given.

Following Baggeroer [1984], Catipovic [1990] discussed the developments of acoustic communication systems up until 1990. Transmission loss modelling techniques had emerged that helped calculate the expected sound fields. The models of the sound field showed that for most underwater environments there was a large variation in the sound intensity, and it was suggested that telemetry designers needed to carefully consider the placement of the transmitters and receivers. They also needed to remain alert to the fact that systems might need to be designed to account for expected failure as the underwater environment changed.

Catipovic [1990] discussed the performance of data transmissions for long-range (typically 20-20,000 km), medium (1-10 km) and short range channels (less than 200 m). Several short range channels had been implemented and it had been concluded that complex communication techniques were not required. For long range deep water channels, however, large pressure signals were required to obtain coherent communication between 200 Hz and 10 kHz. In some cases, coherent communication was not possible due to the degraded quality of the received signal, and incoherent techniques were used to provide a communication system since incoherent techniques can operate in more difficult environments.

Medium range channels were generally shallow water channels. The optimal frequency of operation for shallow water environments was found to be between 10 kHz and 100 kHz. Shallow water environments were found to be particularly difficult environments in which to operate because the principal arrival signal is obscured amongst the signals coming from the many other transmission paths.

Catipovic [1990] noted that synchronisation, equalisation, multi-path processing, modulation, signal design, and coding were active areas of research for underwater acoustic systems at the time. Two particular processes that underwent considerable development in subsequent years were synchronisation and multi-path processing. By performing synchronisation in conjunction with multi-path processing, Stojanovic et al. [1993] were able to achieve phase-coherent communication in more complex environments, particularly shallow water environments. A number of papers that followed [Stojanovic et al., 1994, 1993, 1995] showed how phase coherent communication could be implemented through the design of a receiver structure which jointly updated both the phase-locked-loop (PLL) and the equaliser.

Stojanovic [1996] considered the developments of acoustic communication up until 1996. The work presented by Stojanovic et al. [1993] had enabled coherent communication and thus higher data rates, and was considered a major breakthrough. It allowed for new fields of research to emerge such as distributed networks and designs for autonomous oceanographic telemetry networks. An interesting observation was made concerning the influence of multi-path dispersal of sound on the adaptive equaliser: as the data rate increased, the inter-symbol interference (ISI) increased, requiring greater computational complexity at the receiver; however as the symbol rate is increased, the rate becomes much greater than the rate at which the channel changes, which can allow the adaptive equalisers to perform better.

Stojanovic [1996] noted that to perform phase coherent communication, a system was required to use joint synchronisation and equalisation. Implementing joint synchronisation and equalisation is computationally intensive and methods to reduce this complexity were being investigated. Other areas of research reported as under investigation included real-time implementations, techniques for interference suppression, multi-user systems, self-optimising systems and systems suitable for mobile autonomous underwater devices. Kilfoyle and Baggeroer [2000] and Chitre et al. [2008] have discussed some of these developments, in particular sparse equalisation, blind equalisation, and alternate designs for estimating the phase.

3.2 Channel compensation

In the past few decades, considerable research has been reported on two techniques known as time-reversal and inverse filtering as methods for filtering signals in order to compensate for channel distortions. In this section, a review is made of the development of both of these techniques, and comparisons that have been made between them.

3.2.1 Time-reversal

Early work by Parvulescu

Time-reversal is a technique that involves using the channel through which a signal is emitted to compensate for distortions that result from transmitting through the channel. The process of time-reversal was first performed by Parvulescu in 1961 [Parvulescu, 1995]. Parvulescu was investigating the use of the correlation operator to determine the relationship between a source and multiple receiver locations.

At the time, correlation calculations were computationally expensive and impractical. To reduce the computational requirements, it was proposed that the channel be used to determine the correlations. An impulse was transmitted from a source location into a channel and the signal recorded at a receiver location as $h_1(t)$. The signal, $h_1(t)$, was then played in reverse at the source location. The process of playing a recorded response in reverse is referred to as time-reversal (TR). The signal received as a result of playing the reversed signal and neglecting the noise in the system is given as

$$r(\tau) = \int h_1(-t)h_2(\tau - t).dt \tag{3.1}$$

where $h_2(t)$ is the impulse response of the channel during the second transmission. If the channel response does not change considerably between both measurements, then $h_1(t) \simeq h_2(t)$ and $r(\tau)$ is the auto-correlation of the channel response. If a channel impulse response is sufficiently non-repetitive the auto-correlation function consists of a large spike surrounded by smaller side-lobes.

Parvulescu conducted a number of experiments performing this operation in air in a reverberant room, and also in a shallow underwater environment. In both environments, the response at the receiver location consisted of a large spike, as expected of an auto-correlation function. Experiments in air in the presence of noise demonstrated that the operation continued to result in a spike, even at signal-to-noise ratios of -30 dB.

Parvulescu also examined the temporal and spatial stability of time-reversal in the ocean. The repeated playback of the measured impulse responses in reverse continued to result in spikes being detected for up to eight

Figure 3.1: Phase conjugation holography [Fink, 1992, Fig. 10]

hours after the initial recording. It was also found that variations in the receiver range and depth from the source had an influence on the correlation results. In particular, small variations in the receiver depth were found to cause the peak to vary considerably in magnitude, whilst large variations in range were required to similarly vary the peak magnitude. The size of the region at which the signal is present is known as the focus of time-reversal.

The time-reversal mirror

Time-reversal was also investigated by Prada et al. [1991] as a means of focusing ultrasonic waves for medical applications. The time-reversal technique was simultaneously performed using an array of transducers, which was termed a *time-reversal mirror* (TRM). The application of the TRM to underwater acoustics was examined by Jackson and Dowling [1991], and related to another process known as a *phase conjugate mirror* (PCM).

Time-reversal mirror and phase conjugate mirrors

Phase conjugation involves performing the conjugate reflection of harmonic signals over a surface referred to as the phase conjugate mirror (PCM). The PCM was initially implemented in optics, and Prada et al. [1991] describe a number of techniques used to achieve phase conjugations in that field. One of the methods for performing optical phase conjugation is known as phase conjugate holography. Figure 3.1 shows the method by which phase conjugated holography is performed. The technique involves exposing a photo-refractive film to a wave field, along with a uniform reference wave. After processing the film, the reverse illumination of the film with the reference wave results in the re-emission of the original wave field [Fink, 1992].

When the wave field recorded on the PCM is from a single source, then the re-emitted wave will generate a wave that focuses on the source location [Prada et al., 1991]. The use of phase conjugation in underwater acoustics was investigated by Clay [1966] and Ikeda [1989]. Clay [1966] described the acoustical response with respect to numerous modes of the ocean. As the ocean is dispersive, conventional beam-steering using time delays was found

to be ineffective. Clay therefore used the normal mode model of an underwater environment and proposed that the transmission and reception of a harmonic source can be maximised by adjusting the phase of the transmission array based on the phase of the modes, particularly for the modes having the least attenuation.

Phase conjugation was found to be related to the TRM by both Jackson and Dowling [1991] and Prada et al. [1991]. The TRM is equal to the PCM when the bandwidth of a TRM is reduced to a single frequency. Conversely, phase conjugation is equivalent to time-reversal when simultaneously performed at all frequencies over the bandwidth of operation.

Multi-path compensation

A particular feature of time-reversal is *multi-path compensation* (MPC). Multi-path describes the behaviour of a signal emitted in an environment which exhibits features that cause the signal to reach a target location via a number of different paths, each having a different delay and direction of arrival. The multiple arrivals of the signal cause problems such as fading and inter-symbol interference. Fading is the reduction of the signal level that results from the paths destructively interfering with each other.

A common technique used in signal processing is beam-steering. Beam-steering involves the use of an array of transducers in conjunction with delay taps to emit or receive signals from a single direction. Jackson and Dowling [1991] reported that a TRM effectively performs beam-steering for every path. Multi-path compensation through beam-steering can be explained with reference to the two path system shown in Figure 3.2. In the first step, a short pulse is emitted and the signal arrives at the TRM, arriving from a different direction for each path, with a delay related to the path length. The signal received by each element for a pulse originating from a certain direction will be delayed according to the spatial location of the element, and the direction of arrival of the wave.

In the second step, the signal is played back in reverse on each element of the array. The reversal of the signals for a single pulse received from a specific direction results in the appropriate delay between elements (being the reverse of the received) to steer the pulse in the direction from which it arrived. The time-reversal also allows each pulse received on the array to be re-emitted in such a way that the pulses originating from each path arrive at the original source location at the same time, regardless of the original time gap between them.

Focusing: time-reversal and the wave equation

Prada et al. [1991] were able to show that it was theoretically possible to

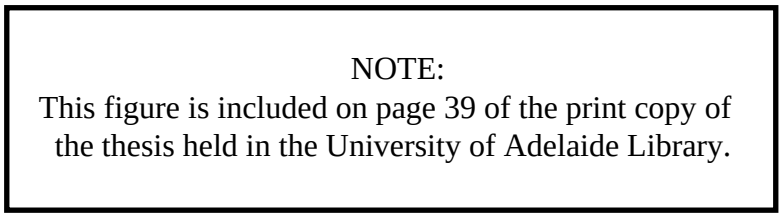


Figure 3.2: Beam steering as performed by a time-reversal mirror [Jackson and Dowling, 1991, Fig. 3]

recreate a sound field through the time-reversal process. Examining the process with respect to the wave equation,

$$\nabla^2 p - \frac{1}{c^2(\mathbf{r})}\frac{d^2}{dt^2}p = 0 \tag{3.2}$$

where p is the pressure field, and $c(\mathbf{r})$ is the speed of sound, then if $p(\mathbf{r}, t)$ is the field for the first part of the time-reversal, it must satisfy the wave equation, and $p(\mathbf{r}, -t)$ must also be a solution of the wave equation since the terms involving time have an even derivative. If $p(\mathbf{r}, t)$ is the pressure field resulting from a point source emitted at $t = 0$ then it is theoretically possible to create an environment with a field defined as $p(\mathbf{r}, T - t)$, such that all the waves will propagate back to the source location, culminating in exciting the point source at $t = T$. It should be noted that if there is no device to absorb all the energy at the source location, then the waves will continue to again propagate outwards.

If the field, $p(\mathbf{r}, t)$, is a solution to the wave equation, then the field, $p(\mathbf{r}, T - t)$, must also be a solution of the wave equation, and thus be physically realisable. The creation of such a field requires that $p(\mathbf{r}, t)$ along with its temporal and spatial derivatives, be recorded at a specific instance. The subsequent field generated with the opposite derivatives initiates field $p(\mathbf{r}, T - t)$. Recording and generating the field and spatial derivative of the field are practically unrealisable as recording can only be done during a finite time interval over a finite region of space. Prada et al. [1991] have shown that when using TRM during a finite time interval and over a finite region of space, the focus continues to be evident even through inhomogeneous media. However, numerical simulations show that the length of the mirror has to be sufficiently large for time-reversal to produce a focal point.

Fink [1992] also addressed the problem of the recreation of an entire pressure field, using Huygens principle. Huygens principle states that the wave field in a volume can be predicted from the field and the normal derivative on a 2-D surface surrounding the volume [Porter and Devaney, 1982]. Thus, time-reversal only needs to be performed on a single surface. When time-reversal is performed on a surface, the enclosed volume is known as a *time-reversal cavity*. Cassereau and Fink [1992] have been able to use time-reversal cavity theory as a means of studying the limitations of a TRM. They note that the field generated by a time-reversal cavity excited by a point source is a field that is the superposition of a converging wave and a diverging wave. The superposition of these two waves results in a limitation of the size of the focal spot to $\lambda/2$ when the excitation signal is a harmonic source with wavelength, λ. Cassereau and Fink [1993] observed that this limitation is transferable to the TRM in that the size of the array has some bearing on the size of the focal region. Jackson and Dowling [1991] examined the performance of a PCM with respect to the size of the array (known as the aperture), and found that the field produced at the source location had an amplitude proportional to the integral of the intensity of the sound received at the array during the recording stage. Thus, in some instances, a smaller array can outperform a much larger array, depending on the intensity of the signal recorded at the array. This significant discovery contradicts the original idea that the focus and compensation of the distortion were dependent on having a large aperture.

Focus steering

Whilst time-reversal has been observed to focus sound at an original source location, it is often desirable to use the recordings made in the time-reversal procedure to reproduce sound at a region away from the initial source location. Several authors have investigated different methods of achieving this.

Dorme and Fink [1996] have studied a method of steering the focal point away from the target zone for an environment that consists of an abberation layer that is between the array and a homogeneous environment. The research used a time-reversal method to excite the environment with a pulse from the array. This pulse passes through the layer and arrives at a reflective object. The object then reflects the wave back through the layer to the array where the pressure signal is recorded.

The recorded signal can then be time-reversed to focus back on the reflector. This form of time-reversal is known as the *pulse-echo mode* of operation. By implementing delays on the signals received from the first stage of the process, similar to that performed in beam-steering, Dorme and Fink found it was possible to shift the focal point away from the original reflector. However the temporal and spatial side-lobes increased as the desired focal

point shifted further from the initial reflector. To improve the focal point when steering, Dorme and Fink developed a method that involves numerically calculating the field on a planar surface on the other side of the layer, based on the signal recorded at the TRM. By applying the delays at this surface and back-propagating the wave form to the array, Dorme and Fink improved the performance of the focal steering. This method was later extended by Tanter et al. [1998] through back-propagation to a curved surface, specifically a human skull, to provide focal steering within the brain.

Shifting of the focal zone within the underwater environment was examined by Song et al. [1998], who developed a technique that used a property of the ocean waveguide to enable the interference structure to be characterised by the existence of lines of maximum intensity having a fixed slope, given by

$$\beta = \frac{r}{\omega}(\frac{\Delta\omega}{\Delta r}) \tag{3.3}$$

where r is the range, and ω is the frequency. This relationship has been termed the *wave-guide invariant*. Numerical simulations that demonstrate this property are shown in Figure 3.3, where it can be observed that the amplitude response is approximately linearly shifted in the frequency domain for a change in range. A similar relationship is also observable for the phase response.

The technique examined by Song et al. [1998] to shift the focal zone uses the wave-guide invariant relationship to estimate the frequency response at ranges surrounding the original focal zone. Figure 3.4 shows simulated results of focal zone shifting for a given environment. The sound intensity is shown for the depths and ranges for standard time-reversal, and time-reversal with a frequency shift of -20 Hz and +20 Hz. The focal zone has been moved 300 m inwards and outwards respectively for each case, having a similar focal structure to the original time-reversal focal zone. Hodgkiss et al. [1999] have presented experimental results that confirm this simulation. Kim et al. [2001a] examined the possibility of using the wave-guide invariant to create nulls at the same time as creating a signal to focus at the focal zone.

Matched field processing

Another area of research that is particularly related to time-reversal is *matched field processing* (MFP). Matched field processing is a source localisation method. The process involves using signals received on an array in an environment to determine the location of a target that is either emitting sound (for passive detection) or is able to reflect sound (for active detection). MFP requires that the environment can be effectively modelled. The model is used in conjunction with the signals recorded on an array to determine the most likely source location of the target. Matched field processing involves

Figure 3.3: Sound intensity for frequencies ranging from 445 Hz to 465 Hz for a simulation of a 140m deep shallow water environment with source and receiver depth of 40 m and 50 m respectively. The curves have been displaced by 2 dB increments for each curve. [Song et al., 1998, Fig 5]

simulating the emission of the time-reversal of the received signals using the model and calculating the sound pressure at various points in the environment. The location displaying the largest magnitude is then considered to be the location of the original source. The first implementation of MFP was by Bucker [1976] who performed MFP using a single harmonic source. Clay [1987] demonstrated that harmonic sources could be replaced with wide-band transmissions of transient or random signals. The position estimate was found to improve dramatically as a result of using a wider bandwidth. Given that the MFP technique essentially uses the time-reversal operator, it can be observed to have the same focal size as using time-reversal in real environments [Kim et al., 2001b].

Focusing improvement by reflections

In section 3.2.1 discussing multi-path compensation, it was shown that a time-reversal mirror effectively transmits the signals received so that they propagate back through the paths through which they came. Dowling and Jackson [1992] investigated this phenomenon and found that media containing a large number of reflectors had a tighter focal point compared to that observed from performing time-reversal in a free-space environment.

Derode et al. [1995] also examined the focal size with and without reflectors for ultrasonic environments. For a linear array operating in a homogeneous media, the smallest size of the focal region was given by the diffraction limit, $\lambda z/a$ where λ is the wavelength, z the distance from the array, and a the size of the array. When scattering was included in the media, the size

Figure 3.4: Sound intensity for range and depth for a time-reversal having an original focal point at a range of 6.2 km and depth of 70 m with (a) no frequency shift, (b) a frequency shift of -20 Hz, and (c) a frequency +20 Hz. [Song et al., 1998, Fig 3]

was reduced to $\frac{1}{6}\lambda z/a$. The improvement in the focal size can be attributed to the fact that each reflector effectively operated as a source, enlarging the virtual aperture of the array. It was also found that when the duration of the reverberation was long, shorter recordings of the reverberation that did not necessarily encompass the initial direct wave could be used to produce a wave that also focused on the target location.

Carsten et al. [1999] considered the case of a single transmitter and receiver in a chaotic cavity. This scenario differs from a time-reversal mirror, particularly as the receiver location is placed within the reverberant environment rather than on the surface. Time-reversal focused the sound at the original location with good spatial and temporal compression. When time-reversal was performed using a variety of finite duration recordings of the response, it was observed that the time-reversal process continued to focus sound at the target location having the same peak magnitude regardless of the time at which the recording started. The duration of the response that was recorded was found to be linearly related to the magnitude of the peak for short durations of recording.

Roux et al. [1997] examined the focusing of time-reversal in a wave-guide that consisted of a media bounded by two surfaces. It was anticipated that such a wave-guide could provide some insight into how time-reversal might perform in a shallow water underwater acoustic environment that was similarly bounded by two surfaces: the air-water interface and the sea floor. The focusing size was found to be increased due to the reflections resulting from the boundaries. By unwrapping the propagation paths due to reflections at the surfaces, the transmission paths could be seen as coming from virtual images of the TRM array, as shown in Figure 3.5. This also demonstrates the idea of increasing the virtual aperture of the array. Roux and Fink [2000] later showed that the number of effective virtual images of the array was limited by the reflectivity of the grazing angle on the boundaries, and also the directivity pattern of the source transmitter. In underwater experiments Kuperman et al. [1998] obtained focal zones at large distances of the order of 100 times that of the aperture of the TRM. These experiments confirmed the assumption that the focusing was improved as a result of the large number of reflections.

Derode et al. [1999] made the interesting observation that the large number of reflections allowed the number of bits used in the time-reversal to be reduced. By reducing the number of bits to a single bit, very low cost, but high energy equipment could be used to generate the signals.

Iterative focusing

Time-reversal has also been investigated as a means of focusing sound energy on a desired target within a human body where it is not possible to place a

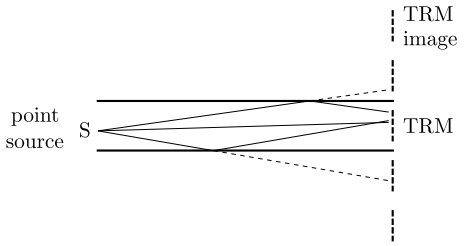


Figure 3.5: Mirror images resulting from a wave-guide [Roux et al., 1997, Fig. 6]

source at the target location, for example, to conduct the acoustic rupture of a kidney stone. When the target is reflective, a probe signal can be used to excite the environment, so that the target reflects a signal back to the array. The signal received at the array can then be time-reversed and transmitted to create an acoustic focus at the target. By boosting the energy during the time-reversal stage, a large acoustic pulse can be sent to the target location whilst at the other locations, the signal should be much smaller. If, however, there are a number of reflective targets in the environment, the time-reversal process actually focuses energy back to each reflector in proportion to the energy reflected from each scatterer. To achieve focusing only on the largest reflector, Ikeda [1989] and Prada et al. [1991] demonstrated that both the phase conjugation and time-reversal methods could be made to focus on the largest scatterer through the iterated application of time-reversal. This operation is shown in Figure 3.6. After the first transmission, the reflective objects return a signal according to their reflectivity. The time-reversal of the received signal at the array can be seen to consist of the summation of a number of signals that will each focus back on the reflective objects. The magnitude of each of these signals is determined by the reflectivity of the object. After emitting the time-reversal signal, the objects will receive a signal according to their reflectivity, and thus reflect a signal having a magnitude of their reflectivity squared. At each iteration, the magnitude of the signal for each receiver in the time-reversal signal is multiplied by the reflectivity of the reflective object. If the time-reversal signal is normalised, then the magnitude of the signal for the object with the largest reflection will remain constant, whilst the magnitude of the signal for the other reflectors will reduce.

The process of iterative time-reversal was also examined by Fink [1992]. Fink found that the iterative process only focused on the strongest reflector when the scatterers where sufficiently spatially separated. In such a scenario, the odd iterations tended to focus on the strongest target whilst reducing the energy at the less reflective targets; but on the even iterations, energy was focused at both targets equally. Prada and Fink [1994] and Prada et al.

NOTE:
This figure is included on page 46 of the print copy of
the thesis held in the University of Adelaide Library.

Figure 3.6: Iterative time-reversal with pulse excitation. [Prada et al., 1991]

[1995] explained this observation by expressing the process as a sequence of matrix operations. Referring to Figure 3.7, the inter-element impulse response between elements m and l shall be denoted as $k_{lm}(t)$, the relationship between the transmitted signals, $e_m(t)$, and the signals received, $r_l(t)$, can then be represented as

$$\begin{bmatrix} r_1(z) \\ r_2(z) \\ \vdots \\ r_m(z) \end{bmatrix} = \begin{bmatrix} k_{11}(z) & k_{12}(z) & \cdots & k_{1m}(z) \\ k_{21}(z) & k_{22}(z) & \cdots & k_{2m}(z) \\ \vdots & \vdots & \ddots & \vdots \\ k_{m1}(z) & k_{m2}(z) & \cdots & k_{mm}(z) \end{bmatrix} \begin{bmatrix} e_1(z) \\ e_2(z) \\ \vdots \\ e_m(z) \end{bmatrix} \tag{3.4}$$

where the signals and impulse responses have been converted to their z-transform equivalent. If the z-transforms are converted to the Fourier equivalent, then for each frequency, the matrix relation can be given by

$$\mathbf{R}^0(\omega) = \mathbf{K}(\omega)\mathbf{E}^0(\omega) \tag{3.5}$$

where the superscript 0 has been added to indicate the initial recording at the time-reversal array. The next set of excitation $\mathbf{E}^1(\omega)$ is then the time-reversal of $\mathbf{R}^0(\omega)$, and is thus given by

$$\mathbf{E}^1(\omega) = \mathbf{R}^{0*}(\omega) \tag{3.6}$$

$$= \mathbf{K}^*(\omega)\mathbf{E}^{0*}(\omega). \tag{3.7}$$

where the super-script $*$ denotes the complex-conjugate operator, being the result of phase-conjugation of the frequency response function. It can then

Figure 3.7: Inter-element impulse response. [Prada et al., 1995]

be observed that for each iteration, the excitation signals for the even and odd iterations is given by

$$\mathbf{E}^{2n}(\omega) = [\mathbf{K}^*(\omega)\mathbf{K}(\omega)]^n \mathbf{E}^0(\omega) \tag{3.8}$$

$$\mathbf{E}^{2n+1}(\omega) = [\mathbf{K}^*(\omega)\mathbf{K}(\omega)]^n \mathbf{K}^*(\omega)\mathbf{E}^{0*}(\omega) \tag{3.9}$$

Many environments are considered to maintain reciprocity. Environments that maintain reciprocity have the same impulse response between two positions regardless of which source the signal originates from. In such environments the matrix $\mathbf{K}(\omega)$ is symmetric, and thus $\mathbf{K}^*(\omega)\mathbf{K}(\omega)$ is Hermetian, and can be diagonalised with orthogonal eigenvectors having positive real eigenvalues. If the initial excitation vector, $\mathbf{E}^0(\omega)$, is decomposed into the eigenvectors of $\mathbf{K}^*(\omega)\mathbf{K}(\omega)$,

$$\mathbf{E}^0(\omega) = \mathbf{F}_1(\omega) + \mathbf{F}_2(\omega) + \cdots + \mathbf{F}_p(\omega) \tag{3.10}$$

then Equations 3.8 and 3.9 result in

$$\begin{aligned}\mathbf{E}^{2n}(\omega) &= \lambda_1^n(\omega)\mathbf{F}_1(\omega) + \lambda_2^n(\omega)\mathbf{F}_2(\omega) \\ &\quad + \cdots + \lambda_p^n(\omega)\mathbf{F}_p(\omega)\end{aligned} \tag{3.11}$$

$$\begin{aligned}\mathbf{E}^{2n+1}(\omega) &= \lambda_1^n(\omega)\mathbf{K}^*(\omega)\mathbf{F}_1^*(\omega) + \lambda_2^n(\omega)\mathbf{K}^*(\omega)\mathbf{F}_2^*(\omega) \\ &\quad + \cdots + \lambda_p^n(\omega)\mathbf{K}^*(\omega)\mathbf{F}_p^*(\omega)\end{aligned} \tag{3.12}$$

where λ_i, $i \in [1, p]$ are the eigenvalues of $\mathbf{K}^*(\omega)\mathbf{K}(\omega)$ with $\lambda_1 > \lambda_2 > \cdots > \lambda_p$. It is worth noting that the eigenvalues of $\mathbf{K}^*(\omega)\mathbf{K}(\omega)$ are the square of the singular values of $\mathbf{K}(\omega)$. After a number of iterations of the process, n is large, and thus

$$\mathbf{E}^{2n}(\omega) \simeq \lambda_1^n(\omega)\mathbf{F}_1(\omega) \tag{3.13}$$

$$\mathbf{E}^{2n+1}(\omega) \simeq \lambda_1^n(\omega)\mathbf{K}^*(\omega)\mathbf{F}_1^*(\omega) \tag{3.14}$$

demonstrating the different results for the odd and even iterations. If the environment is modelled as an environment consisting of multiple point scatterers with no inter-scatterer reverberation, Prada et al. [1995] have shown that the matrix $\mathbf{K}(\omega)$ can be given by

$$\mathbf{K}(\omega) = \mathbf{H}(\omega)^{\mathrm{T}}\mathbf{C}(\omega)\mathbf{H}(\omega) \tag{3.15}$$

where $\mathbf{H}(\omega)$ is the transfer matrix describing the response between each element and each scatterer, and $\mathbf{C}(\omega)$ is a diagonal matrix containing the reflectivitys of the scatterers.

For well-separated targets, the eigenvectors of $\mathbf{K}^*(\omega)\mathbf{K}(\omega)$ are related to the vectors to focus on each scatterer individually. Under such conditions, it is possible to focus on individual scatterers by transmitting the different eigenvectors of the time-reversal operator. The technique is called DORT (French acronym for *Decomposition of the time-reversal operator*). A thorough description and analysis of the process is given by Prada et al. [1996]. Decomposition is particularly effective when many iterations of time-reversal are required to focus on the strongest target due to similar reflectivity's.

In some situations, such as symmetric environments, where the eigenvectors do not correspond to each target in a one-to-one relationship, the DORT failed. However, it was found that by transmitting combinations of eigenvectors the targets could be targeted separately. The assumption that each scatterer results in a single eigenvector was shown to be false by Chambers and Gautesen [2001], where it was theoretically shown that spherical scatterers can have up to four eigenvectors. However, when the reflectors are made from hard material, only a single eigenvector dominates.

Mordant et al. [1999] examined the eigenvectors with respect to frequency and observed that the singular values corresponding to each reflector can often be related between frequencies. Using such relationships, wide-band signals can also be used to target each reflector.

Several authors have extended the eigenvalue iteration technique:

- Prada and Fink [1998] examined DORT for an air filled cylinder in water. The DORT method was used to isolate waves known as Lamb waves that travel around the cylinder. These waves resulted in the observation of two eigenvalues for the clockwise and anti-clockwise propagation.

- Mordant et al. [1999] examined the performance of the DORT method for a scatterer as it moved close to an interface having a reflectivity close to -1 (as is the case for a water/air interface). Under such conditions, the reflection from the scatterer and the surface was found to be hard

to resolve due to the reflected and direct waves cancelling each other out. The distance at which the scatterer was no longer detectable was $\lambda/5$ at a range of 400λ.

- Kim et al. [2001a] used the concept of the wave-guide invariant (see Equation 3.3) to alter the time-reversal process to create null locations in the underwater environment. Using this technique it was possible to selectively focus on two targets and obtain identical eigenvectors as would be obtained from the DORT technique.

- Lingevitch et al. [2002] altered the DORT technique used in the ocean to use the entire array to transmit the initial excitation for underwater environments to achieve better excitation when locating targets.

- Kerbrat et al. [2003] used the DORT technique as a means to find cracks within material. The DORT technique in general outperformed the transmit/receive focusing and also a time-reversal technique.

Time-reversal and the matched filter

Time-reversal is closely related to matched filtering, described in a previous subsection. A matched filter is formed from the time-reversal of the transmission symbol. However, time-reversal differs from matched filtering due to the fact that the entire channel response is time-reversed and used as a filter. Fink [1992] notes that matched filtering is inherently different from time-reversal since time-reversal involves performing the time-reversal at the transmitter, and allowing the ocean to perform the convolution. Using time-reversal at the transmitter results in the focusing of the sound at the receiver.

Time-reversal may also be implemented at the receiver, as done by Dowling [1994]. When time-reversal is implemented at the receiver, it is often called a *passive phase conjugate filter*. The implementation involves measuring the channel response at the receiver and using the time-reversal of the response as a filter for subsequent communication signals. Dowling [1994] showed that the filter was able to provide vast improvements over matched filtering.

There is some confusion over the term 'matched filter' however, as communication theory textbooks sometimes refer to it as the filter obtained from time-reversing the general symbol waveform; whilst some authors, such as Clay [1987], Li and Clay [1987] and Kuperman et al. [1998] have used the term 'matched filter' to refer to the signal that results from the time-reversal of the wave-guide response. Of particular note is that Dowling [1994] who introduced the passive phase conjugation process compared time-reversal of

the wave-guide with the matched filter (described as the time-reversal of the pulse waveform).

Another distinction between time-reversal and matched filtering is that time-reversal is generally implemented without consideration for the noise spectrum. In communication theory, the matched filter is often implemented along with a noise-whitening filter. It is interesting to note that the paper by Clay [1966], being one of the earliest papers on phase conjugation, takes into account the noise spectra. Clay observed that if the amplitude of the source excitation of each mode was a_m and the array response was U_m and the noise spectrum for that mode was N_m, then the gain at the receiver, b_m, that maximises the signal-to-noise ratio for the reception of that mode is given by

$$b_m = \frac{a_m^* U_m^*}{N_m^2} \tag{3.16}$$

This equation is similar to a matched filter ($b_m = a_m^* U_m^*$) as described by Proakis [2001] with the addition of a noise-whitening filter, $\frac{1}{N_m^2}$.

Time-reversal and ocean acoustics

In the subsection on underwater acoustics, the propagation of acoustic waves in the ocean was introduced. The propagation of acoustic waves is largely influenced by the sound speed profile and the surface and sea-floor characteristics. The current study is focused on the shallow water environment which is a highly reverberant environment where sound propagates via many reflections with the sea surface and the sea floor, resulting in many paths, a process known as multi-path transmission. Multi-path transmission lengthens the duration of the ocean response between two locations which makes analysis of this environment computationally expensive. An example of the duration of the ocean response versus range is given in Sabra et al. [2002] where the duration of the response for a distance of 10 km was 0.1 second. Time-reversal presents a method to compensate for the long duration using a low-complexity method. However, time-reversal has been formulated for environments having a fixed sound-speed profile, whereas in the underwater environment, the sound speed profile changes with time. Kuperman et al. [1998] showed that time-reversal could still be used for short periods of time, over which sound-speed can be assumed to remain sufficiently static.

The earliest known work that involved using time-reversal in the underwater environment was performed by Parvulescu in 1961. The experiments conducted demonstrated time-reversal working with a single source and receiver. Following this initial work, several authors [Jackson and Dowling, 1991, Roux et al., 1997] described theoretical concepts associated with the use of time-reversal in underwater acoustics, however it was not until 1996 that Kuperman et al. [1998] performed time-reversal experiments in the ocean.

The experiments conducted by Kuperman et al. [1998] examined the focal zone of the time-reversal process, and the stability of the focal zone for extended durations. A second set of experiments were performed in 1997 to investigate further developments on the TRM applied in underwater acoustics. The results of these experiments are presented by Hodgkiss et al. [1999].

Many of the underwater acoustics experiments conducted by Kuperman et al. [1998] investigated the advances of time-reversal that had been developed in ultrasonic research. It is of interest to note that the distance between the TRM and the focal point in ultrasonics was an order of half the size of the aperture of the array, whilst in contrast, the underwater experiments conducted by Kuperman et al. examined the focus at distances of around 100 times the aperture of the TRM [Hodgkiss et al., 1999]. The size of the focal zone at 30 km was observed to be 25 m high, and estimated to be 800 m wide. The ability to focus at such great ranges was attributed to images that are formed due to the reflections at the sea-surface and sea-floor. The duration which time-reversal continued to focus was found to depend on the fluctuation of the sound-speed profile at the focal location. In one of the experiments, time-reversal targeting a depth of 81 m continued to create a focal zone after 10 days, whilst for another case targeting a depth of 47 m the focal zone had degraded considerably after 15 minutes, presumably because the sound speed profile is more stable at the sea floor.

Kuperman et al. [1998] described the focusing of time-reversal in the ocean through the use of normal modes and ray tracing theory. Whilst the discussion with respect to normal modes was based on a harmonic solution of a PCM, the solution can be extended to a broadband application to understand the TRM. If an array spans the entire water column, then all modes are excited, and the field due to a PCM can be approximated by [Kuperman et al., 1998, Eq. 9]

$$P_{pc}(r, z; \omega) \simeq \sum_m \frac{u_m(z) u_m(z_{ps})}{\rho(z_{ps}) k_m \sqrt{rR}} \exp(ik_m(r - R)) \tag{3.17}$$

where $P_{pc}(r, z; \omega)$ is the pressure field at depth, z, and range, r, from the array for the source frequency ω. In this equation, $u_m(z)$ are the mode shape functions, z_{ps} is the depth of the probe source, $\rho(z)$ the density function, k_m the modal wave number and R the distance between the probe source and the PCM. At the focal range ($r = R$), the exponential term equates to 1, and the remaining portion is a scaled spatial correlation function having a peak at $z = z_{ps}$. By summing over the modes, the peak is reinforced resulting in a stronger focus and reduction in the side-lobes. The vertical size of the focal zone can then be related to the mode having the smallest vertical wave length. The vertical size of the focal point can be roughly estimated by depth divided by the wavelength of the highest order mode. Figure 3.8 shows results

Figure 3.8: Sound intensity for phase conjugate (single frequency) mirror from a simulation for a probe source located at a depth of 40 m and range of 6.3 km in a shallow underwater acoustic environment. [Kuperman et al., 1998, Fig. 4b]

from a simulation that demonstrates the strong vertical focus that occurs at the source location. Roux and Fink [2000] showed that the highest effective mode is dependent on the attenuation with respect to the grazing angle, and the spacing of the elements of the TRM contributed substantially to the side-lobes present in the response. The influence of the bottom attenuation was confirmed by Kim et al. [2001b], who termed it *mode stripping*.

The ability for time-reversed signals to maintain a focus for considerable time from the initial excitation can be attributed to the stability of the mode shapes. As the sound speed profile is known to vary considerably at the surface, so also are the mode shapes subject to variability at the surface, whilst at lower depths, the modes can be considered more stable, thus explaining the difference in the temporal stability of the focusing for the 81 m depth and 47 m depth.

3.2.1.1 Further developments of time-reversal

Further developments that have been made involving time-reversal in the ocean include:

- Song et al. [1999] examined the DORT iteration process in the ocean. The technique was found to provide minor spatial focusing improvements. It was found that for multiple scatterers, the reflecting strength of the scatterer alone did not determine which scatter would be the focus. The environment also had an influence. The iterations of the time-reversal procedure also narrowed the bandwidth of the signals to the most effective frequencies.

- Rose et al. [1999] showed that time-reversal mirrors continue to work when used in the underwater environment, regardless of the surface wave height. A technique was proposed that could determine the height of the surface waves from the TR and a measure of the surrounding field. The technique was validated in an ultrasonic experiment.

- Whilst examining matched field processing, Yoo and Yang [1999] observed that some of the modes coupled between the transmitter and receiver array are distorted through the internal waves in the ocean. By eliminating these modes, a technique was developed that improved the likelihood of determining the target location at the expense of a reduction in the spatial resolution.

- Performance of TR for noisy (or low signal level) environments was examined by Sabra et al. [2002]. The signal to noise ratio at the focal point was found to be related to the signal bandwidth and the duration of the signal pulse. Time-reversal was found to reject noise better in reverberant environments than in free-space due to the multi-path.

- Sabra and Dowling [2004] developed a method that was able to perform blind deconvolution in an ocean environment. The technique used the spatial diversity of an array to obtain estimates of the Greens functions due to an unknown source transmission, and then utilise these Greens functions in conjunction with time-reversal and a non-regularised inverse filter to determine the original transmission.

Time-reversal has also had considerable development in medical research. Fink et al. [2003] provides a useful overview of such research. Of particular interest:

- Time-reversal has been used as a means of improving lithotripsy, being the non-invasive damaging of kidney stones through focused high-intensity acoustic pulses. Thomas et al. [1996] showed that time-reversal can be used to move the focus and track the stone during lithotripsy treatment.

- Time-reversal has been used in a process known as ultrasonic medical hyperthermia which is a form of brain therapy. Ultrasonic medical hyperthermia involves exciting body tissue with high intensity ultrasound so that it is absorbed and converted to heat, destroying the cancerous tissue. Tanter et al. [1998] investigated the use of time-reversal focusing to steer through the skull, and Pernot et al. [2004] tested the method using sheep skulls. Time-reversal improved the focusing, and also provided the ability to steer the signal up to 2 cm away from the initial focal point.

- Time-reversal has also been used to improve ultrasonic imaging, through the use of an environment containing scatter media to increase the effective aperture. A draw-back compared to conventional beam-steering techniques is that the entire field needs to be mapped [Roux et al., 2000].

Time-reversal has also been examined for application to solids. Time-reversal in solids is somewhat different to that in fluids as both longitudinal and transverse waves result from the excitation of the media [Draeger et al., 1997]. Several researchers [Kerbrat et al., 2002, Leutenegger and Dual, 2002, 2004, Park et al., 2007, Goursolle et al., 2008] have investigated the use of time-reversal in solids to perform non-destructive testing to locate cracks and air gaps in a variety of scenarios.

3.2.2 Inverse filtering

History

In the late 1970s, several researchers investigated the ability to replicate a desired sound at the ears of a listener sitting some distance from a set of loudspeakers. The acoustic waves propagating from the loudspeakers to the listener are generally distorted as a result of the speaker characteristics and the reflections from the room. In a two-speaker scenario, where it is desired that the sound from each speaker is heard only at the corresponding ear (i.e. left speaker for left ear, right speaker for right ear), the sound that each ear hears from the opposite speaker is known as cross-talk. Researchers have investigated systems to compensate for the distortion and reduce the cross-talk to perfectly reproduce a recorded signal. A related problem is being able to record audio from a source in a reverberant environment using multiple microphones. Both of these problems require systems to be designed that can compensate for distortion resulting from sound propagation in a room: in the former case, the compensation is performed prior to the transmission of sound; and in the later case, the compensation is performed after the reception of the sound.

The designs employed by Flanagan and Lummis [1970], Damaske [1971] and Allen et al. [1977] to compensate for the signal distortion (or the cross-talk), pass the signals through phase shifters. A more thorough method of achieving compensation is to pass the signals through filters that alter both the phase and amplitude of the signals to perfectly compensate for the channel response and cancel out the cross-talk. Such a filter is known as an inverse filter.

The schematic for the two designs of the inverse filter are shown in Figure 3.9. Figure 3.9a illustrates the scenario when a pre-recorded stereo sound

is being reproduced and Figure 3.9b illustrates the scenario for the perfect recording of a sound source. The mathematical description of the multi-channel inverse filtering problem is to find a set of filters, $h_{i,j}(t)$, for an environment with propagation paths, $c_{j,i}(t)$. From Figure 3.9a, it can be observed that signal, $r_j(t)$, reproduced at ear $j \in [1,2]$ is given by

$$r_j(t) = \sum_i t_i(t) \star c_{j,i}(t) \tag{3.18}$$

where $t_i(t)$ is the signal transmitted by speaker i and $\star$ is the convolution operator. If the signals, $t_i(t)$, emitted by the speaker are created as filtered versions of the recorded signals, $s_j(t)$, then

$$t_i(t) \;=\; \sum_j s_j \star h_{i,j}(t). \tag{3.19}$$

Equation 3.18 and 3.19 can be presented in matrix notation,

$$\begin{aligned}
\begin{bmatrix} r_1(t) \\ r_2(t) \\ \vdots \\ r_J(t) \end{bmatrix} \;=\; & \begin{bmatrix} c_{1,1}(t) & c_{1,2}(t) & \cdots & c_{1,K}(t) \\ c_{2,1}(t) & c_{2,2}(t) & \cdots & c_{2,K}(t) \\ \vdots & \vdots & & \vdots \\ c_{J,1}(t) & c_{J,2}(t) & \cdots & c_{J,K}(t) \end{bmatrix} \\
& \star \begin{bmatrix} h_{1,1}(t) & h_{1,2}(t) & \cdots & h_{1,J}(t) \\ h_{2,1}(t) & h_{2,2}(t) & \cdots & h_{2,J}(t) \\ \vdots & \vdots & & \vdots \\ h_{K,1}(t) & h_{K,2}(t) & \cdots & h_{K,J}(t) \end{bmatrix} \\
& \star \begin{bmatrix} s_1(t) \\ s_2(t) \\ \vdots \\ s_K(t) \end{bmatrix}
\end{aligned} \tag{3.20}$$

where $\star$ represents the matrix-wise convolution operator.

In a similar fashion, the matrix representation for the impulse response function between a sound source, $s_1(t)$, and the output of the filters, $r_1(t)$, for the recording scenario shown in Figure 3.9b is given by

$$\begin{aligned}
[r_1(t)] \;=\; & \begin{bmatrix} h_{1,1}(t) & h_{1,2}(t) & \cdots & h_{1,J}(t) \end{bmatrix} \\
& \star \begin{bmatrix} c_{1,1}(t) \\ c_{2,1}(t) \\ \vdots \\ c_{J,1}(t) \end{bmatrix} \star [s_1(t)]
\end{aligned} \tag{3.21}$$

Representing the matrices as $\mathbf{H}(t)$, $\mathbf{C}(t)$, the purpose of inverse filtering is to find $\mathbf{H}(t)$ such that

$$\mathbf{C}(t) \star \mathbf{H}(t) = \mathbf{\Delta}(t) \tag{3.22}$$

for the sound reproduction scenario, and

$$\mathbf{H}(t) \star \mathbf{C}(t) = \mathbf{\Delta}(t) \tag{3.23}$$

for the recording scenario, where

$$\mathbf{\Delta}(t) = \begin{bmatrix} \delta(t) & 0 & \cdots & 0 \\ 0 & \delta(t) & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \delta(t) \end{bmatrix} \tag{3.24}$$

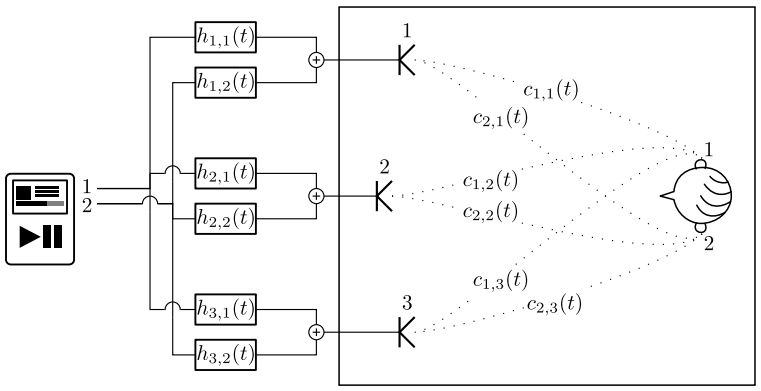
and $\delta(t)$ is the Dirac delta function.

Neely and Allen [1979] showed that for certain rooms, the filter model for a single input / single output system was non-minimum phase, meaning that a stable exact inverse filter cannot be realised. However, by using multiple transmissions for a single output, Miyoshi and Kaneda [1988] first showed that an exact inverse filter could be achieved using multiple transmitters and receivers in a technique that was called MINT. In order to be able to perform the inverse filtering *in-situ*, Nelson et al. [1992] developed a means of determining the multi-channel inverse filters using an adaptive LSE (Least Square Error) method to perform both an inverse filter and cross-talk cancellation. Both the MINT and LSE methods were shown by Nelson et al. [1995] to result in the same co-efficients when the system channel responses were minimum phase.

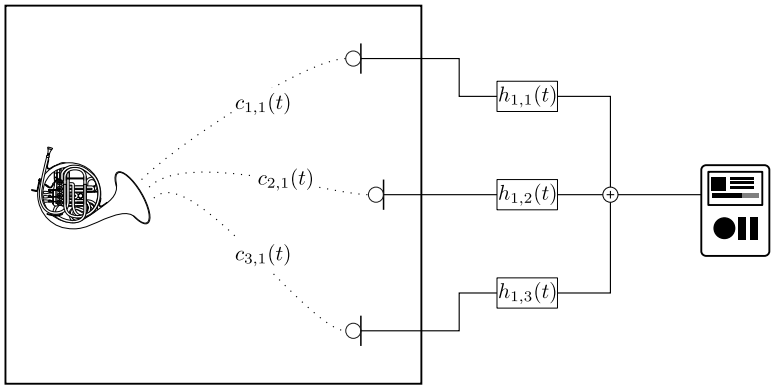
Fast inverse filter design using FFT - the Tikhonov inverse filter

The direct inversion of measured IRFs (Impulse Response Functions) using time domain techniques (see for example Nelson et al. [1995]) are particularly complex and require considerable computational effort. To speed up the calculations, Kirkeby et al. [1996a] developed a method that reduced the computational effort required to design the inverse filter by performing the inversion within the frequency domain. The technique involves the use of a *regularisation parameter* to ensure causality so that wrap-around does not occur on the conversion back to the time domain.

The design of the Tikhonov regularised inverse filter is based on the system presented in Figure 3.10. A set of signals, $\mathbf{s}(z)$, are transformed by the filter, $\mathbf{A}(z)$, to produce a set of signals, $\mathbf{d}(z)$, that are to be replicated by the signals, $\mathbf{r}(z)$, being the output of the electro-acoustic system denoted by $\mathbf{C}(z)$. In order to achieve this, a filter, $\mathbf{H}(z)$, is designed given that $\mathbf{C}(z)$ and



(a) Sound Reproduction



(b) Sound Recording

Figure 3.9: Room configurations for the application of inverse filtering.

$\mathbf{A}(z)$ are known. When applied to the transmission signals, $\mathbf{s}(z)$, the filter produces another set of signals, $\mathbf{t}(z)$, that, when played through the channel $\mathbf{C}(z)$ result in the signals $\mathbf{r}(z)$ at the receivers.

Often the transfer matrix, $\mathbf{A}(z)$, is a delay to ensure causality, i.e. $\mathbf{A}(z) = z^{-m}\mathbf{I}$, or in the case of a communication system, the channel spectral shaping filter response, $\mathbf{A}(z) = g(z)\mathbf{I}$. This problem can be expressed as

$$\mathbf{r}(z) \quad = \quad \mathbf{C}(z)\mathbf{t}(z) \tag{3.25}$$

with the objective that

$$\mathbf{r}(z) \quad = \quad \mathbf{A}(z)\mathbf{s}(z). \tag{3.26}$$

Given that

$$\mathbf{r}(z) = \mathbf{C}(z)\mathbf{H}(z)\mathbf{s}(z), \tag{3.27}$$

the filter $\mathbf{H}(z)$ is designed so that $\mathbf{C}(z)\mathbf{H}(z)$ approximates $\mathbf{A}(z)$. Kirkeby et al. [1998] proposed a cost function to achieve this, along with a term to

regulate the energy of the transmitted signal. The cost function is given by

$$J(z) = \mathbf{e}^{\mathrm{H}}(z^{-1})\mathbf{e}(z) + \kappa \mathbf{t}^{\mathrm{H}}(z^{-1})\mathbf{t}(z) \tag{3.28}$$

where $\mathbf{e}(z) = \mathbf{d}(z) - \mathbf{r}(z)$ is the error signal, and κ is a weighting term applied to the energy of the transmitted signal known as the *regularisation parameter*. The solution to this equation is given by [Kirkeby et al., 1998]

$$\mathbf{H}(z) = \left(\mathbf{C}^{\mathrm{H}}(z^{-1})\mathbf{C}(z) + \kappa \mathbf{I}\right)^{-1} \mathbf{C}^{\mathrm{H}}(z^{-1})\mathbf{A}(z) \tag{3.29}$$

and its frequency domain equivalent,

$$\mathbf{H}(\omega) = \left(\mathbf{C}^{\mathrm{H}}(\omega)\mathbf{C}(\omega) + \kappa \mathbf{I}\right)^{-1} \mathbf{C}^{\mathrm{H}}(\omega)\mathbf{A}(\omega). \tag{3.30}$$

which was observed by Kirkeby et al. [1998] to be the Tikhonov regularised inverse filter design. An extensive discussion of the Tikhonov inverse of a matrix can be found in Hansen [1998].

Figure 3.10: Generic inverse filter system schematic [Kirkeby et al., 1998].

Kirkeby et al. [1998] observed that if the regularisation parameter, κ, was large enough then the temporal wrap-around was negligible, allowing a causal filter to be calculated in the frequency domain using the fast Fourier transform. Calculation of the filter by this technique proved to be considerably faster than equivalent calculations performed using time domain techniques.

Other methods of obtaining an inverse filter

An alternative approach to obtaining the inverse filter solution has been developed by Montaldo et al. [2004], where an approximation of the inverse filter may be obtained experimentally through iterating a time-reversal technique. This iterative method was developed for use in an ultrasound application where it was found that iterative time-reversal was faster than performing any direct calculation of the inverse filter. For underwater acoustic communication however, the transmission times are much longer and the iterative

technique becomes impractical due to the long propagation time within the ocean. However, the iteration could be implemented in software as discussed by Higley et al. [2006], who showed that when implemented in software the technique was mathematically equivalent to the Neuman matrix inverse approximation.

Applications of inverse filtering

Some of the applications for inverse filtering that have been investigated include:

- A number of authors investigated the use of inverse filtering to replicate a plane-wave acoustic field for a 2-D surface using various configurations of discrete sources surrounding the surface in a free field environment. In particular, Kirkeby and Nelson [1993] found that the size of the array, and the angle between the sources with respect to the 2-D surface were critical to providing a good replication of the desired field; Nelson [1994] described various signal processing techniques including the inverse filter to achieve sound reproduction, and Kirkeby et al. [1996b] showed good reproduction of a sound field using only a few loudspeakers.

- Nelson et al. [1995] and Kim and Nelson [2003] examined the spatial extent of the zone of equalisation. The zone of equalisation was found to be related to both the wavelength of the maximum frequency and also the arrangement of the sensor array.

- Kim and Nelson [2004b] examined the influence of the geometrical arrangement of the microphones and speaker on the condition number. An optimally arranged sensor array was developed that was far superior to that of a planar array spanning equivalent dimensions.

- Kim and Nelson [2004b] compared two techniques to obtain an appropriate regularisation parameter. The techniques compared were the General Cross-validation (GCV) method developed by Golub et al. [1979], and the L-curve method described by Hansen [1998] for solving matrix inverse problems. Neither method was found to be the best as each method was suited to different environmental conditions.

- A number of authors investigated the use of inverse filtering to determine the source strength of acoustic sources. Nelson and Yoon [2000] investigated the conditioning of the inverse problem with regard to the geometry and the number of sources and measurement positions. Nelson and Yoon found that the inverse problem became badly conditioned when the wavelength of the radiated sound becomes large

compared with the distance between the sources. However altering the position of the measurement points improved the conditioning of the system being the result of small singular values. Kim and Nelson [2004a] examined the estimation of acoustic source strength in a cylindrical duct. The small singular values were found to relate to the evanescent modes. The conditioning of the system could be increased by locating the measurements close to the acoustic sources in order to increase the singular values related to the evanescent modes.

- Kim et al. [2006] examined the use of inverse filtering to perform cross-talk cancellation for multiple listeners and also examined the robustness to head movement. A system was examined that used four sources and four receiver locations (i.e. two listeners). Source locations were chosen along an array that resulted in the smallest condition number in different frequency bands. Simulations showed that it was possible to achieve cross-talk cancellation. It was found that the frequencies having a well conditioned transfer matrix had a larger spatial extent and less ringing than those having an ill-conditioned transfer matrix.

3.2.3 Comparisons between time-reversal and inverse filtering

Both time-reversal and inverse filtering techniques have been compared on a number of occasions. Clay and Saimu [1988] used a deconvolution in matched field processing, which is similar to inverse filtering without regularisation. The deconvolution resulted in a high frequency resonance that was eliminated using filtering. The MFP results showed that the deconvolution method gave fewer false source locations when compared to using time-reversal; however, the deconvolution was found to be less robust at instances when the inverse filters performed poorly, resulting in ringing that could not be eliminated.

As described in an earlier subsection, the ability for time-reversal to provide focusing assumes that the medium is loss-less [Dorme and Fink, 1995]. However, Thomas and Fink [1996] desired to perform time-reversal through a human skull which consists of a lossy media. Experiments were performed that incorporated compensation for the amplitude variations on each transducer induced by an abberation layer within the environment. The compensation involved applying a gain on each transducer that matched the attenuation observed when comparing the homogeneous media (water only) to the inhomogeneous media, thus effectively resulting in an inverse filter. The results are shown in Figure 3.11. Thomas and Fink [1996] proposed that the method could be improved by performing amplitude compensation for each frequency. The method was also employed by Tanter et al. [1998] to

Figure 3.11: Improved focusing obtained through the use of time-reversal in conjunction with amplitude compensation [Thomas and Fink, 1996].

steer the focus away from its main target and also discussed in greater detail by Tanter et al. [2000].

Cazzolato et al. [2001] performed a comparison between time-reversal and Tikhonov inverse filtering in order to produce a pulse in a simulation of a 145 m deep shallow water at ranges between 2 and 5 km. By using the Tikhonov regularised inverse filters, Cazzolato et al. [2001] achieved greater spatial and temporal focusing than time-reversal. The performance improvement obtained using the Tikhonov regularised inverse filters over time-reversal was attributed to the impulse response of the time-reversal system being similar to the auto-correlation of the impulse response of the channel. The similarity arises because time-reversal uses the time-reversal of the channel response as a filter for the channel. Similarly, the autocorrelation of a channel response can be calculated by the convolution of the time-reversal of the channel response with the channel response itself. The difference between the auto-correlation and the time-reversal process impulse response is that the channel in the time-reversal process differs from that which is used to design the filter and the channel response may also generate additional noise. The frequency response of the auto-correlation of a channel is the frequency response of the channel squared. The squaring operation results in a positive definite frequency response and thus where there is destructive interference there will be significant "dropouts". By using time-reversal with an array, these dropouts can be reduced by ensuring that the destructive interference is not at common frequencies for each channel response between the transmitter and receiver locations. However, this cannot always be ensured and destructive interference can still occur. The flatness of the system frequency response is

thus dependent on all the time-reversal responses averaging out to provide a flat frequency response. In contrast, the Tikhonov inverse filter uses a cost function to achieve a flat response provided it does not consume too much power to do so.

Yon et al. [2003a] examined and compared the time-reversal process with a spatio-temporal inverse filter to focus sound in rooms using a loudspeaker array. Whilst previous work presented in Yon et al. [2003b] had shown that time-reversal is able to create a focal zone, the loss of information during the time-reversal process was considered to degrade the quality of the focus. A spatio-temporal inverse filter based on singular value discarding was investigated as a means to improve the focusing. It was found that the spatio-temporal inverse filter had better spatial focussing compared to the time-reversal filter, provided that the bandwidth of the signal was not too small, at which point they become similar to each other. In addition, the temporal focusing of the spatio-temporal inverse filter far exceeded that of time-reversal, with the temporal side-lobes of the signal at focus being almost 20 dB lower than for time-reversal. It was also shown that the spatio-temporal inverse filter was able to provide control over a spatial sound field, effectively using multiple control points over which to optimise the inverse filter.

The relationship between time-reversal and inverse filtering methods was investigated by Vignon et al. [2006]. Their work investigated the relationship between the time-reversal filter and the spatio-temporal inverse filter. Previous publications had shown that the time-reversal array was required to completely surround the medium desired to be controlled (forming a time-reversal cavity) to avoid echoing, whereas an inverse filter is able to avoid echoing using an array that does not completely surrounded the medium to be controlled. To examine this phenomenon, a relationship was formed between the time-reversal and the inverse filter for a system comprising of two arrays located either side of a solid interface submersed in water. It was shown that the set of signals, E_{IF}, that result from using the inverse filter to transmit from array 1 to focus on a target transducer, S_2, located on array 2 is given by

$$E_{\mathrm{IF}} = \mathbf{H}^* S_2 + \mathbf{H}^{-1}\mathbf{K}_2\mathbf{K}_2^* S_2, \tag{3.31}$$

where $\mathbf{H}$ is the transfer matrix between the elements of array 1 and array 2, and $\mathbf{K}_2$ is the transfer matrix between array 2 and itself. The relationship between the inverse and time-reversal filter was shown by noting that the signal, E_{IF}, is the sum of the signal resulting from a time-reversal procedure between array 1 and 2, $\mathbf{H}^* S_2$, and the signal resulting from the time-reversal procedure between array 2 and itself, $\mathbf{K}_2\mathbf{K}_2^* S_2$, multiplied by $\mathbf{H}^{-1}$ to account for the signals being emitted from array 1 instead of array 2. This result demonstrates that the use of an inverse filter on a single array is equivalent

to using time-reversal on both arrays simultaneously.

3.3 Channel compensation techniques used in acoustic communication systems

Although time-reversal (TR) was demonstrated in 1961 by Parvulescu and Clay [1965], the earliest reference found that proposes the use of the technique with communication systems was given by Jackson and Dowling [1991]. Several other authors suggested the use of time-reversal to assist communication [Kuperman et al., 1998, Hodgkiss et al., 1999, Kim et al., 2001a].

The implementation of time-reversal has been investigated using two different techniques: active and passive time-reversal (commonly referred to as passive phase conjugation in the literature). These techniques are shown in Figure 3.12a and Figure 3.12b respectively. The steps for active time-reversal communication are:

1. The target transmits a pulse, and the transmitting array records the pulse at each element in the array.

2. The pulses recorded at each element are used as a filter between the data signal and the signal to transmit at each element.

The steps involved in passive time-reversal communication systems are:

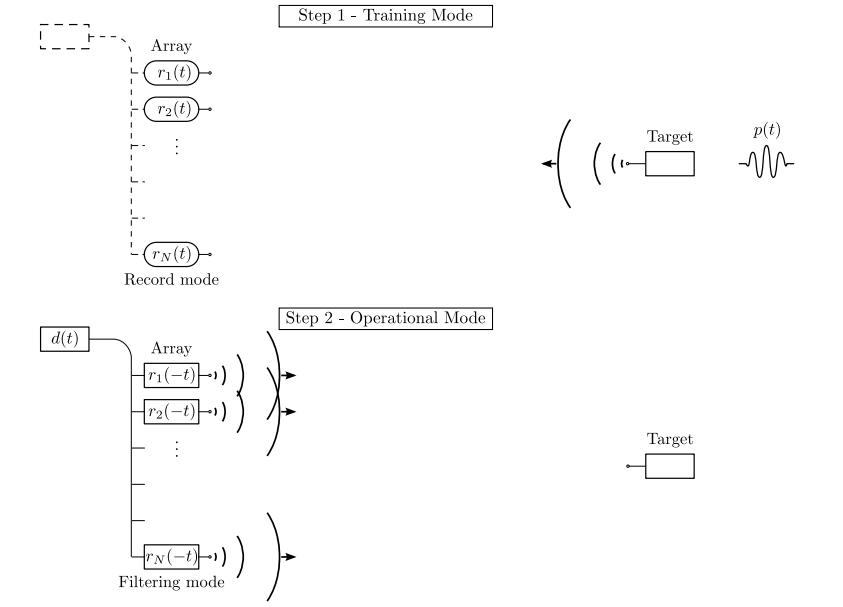
1. Transmitter source emits a single pulse, followed by a delay (in which the response at the receiver has had time to decay away), followed by the data to be transmitted.

2. The receiver captures the response from the first pulse, then uses the time-reversal of this signal as a filter for the future signals that are transmitted. Often the receiver consists of an array whereby the outputs of the filters on each element of the array are combined.

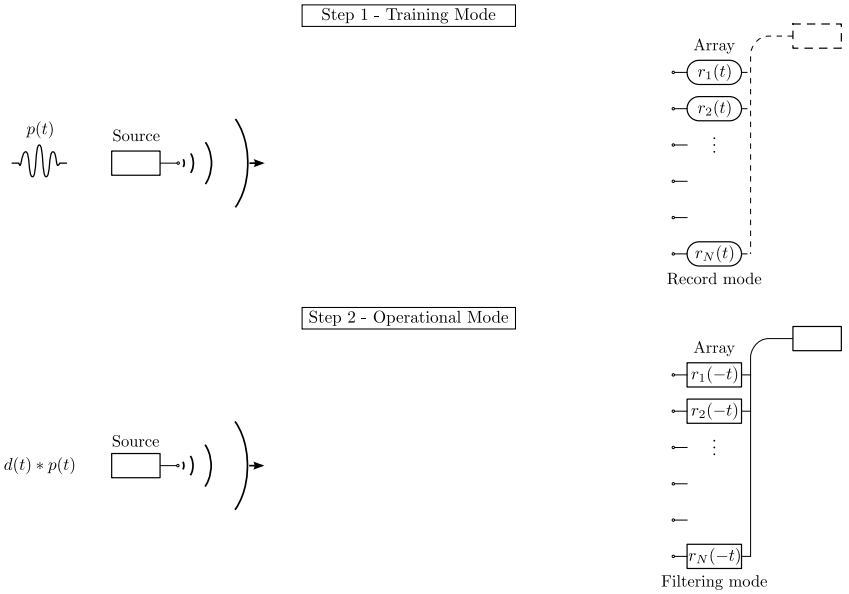
Passive time-reversal can be seen to be advantageous for scenarios where it is too costly, or not feasible to have a transmitter at both ends of the transmission system. However, the advantage of the active implementation over the passive implementation is that the active implementation actually results in a spatial focusing of the signal at the receiver.

3.3.1 Passive time-reversal in underwater acoustic communication

The earliest reference found for the implementation of time-reversal was by Dowling [1994] who used the passive-phase conjugation technique in a deep



(a) Active time-reversal communication system.



(b) Passive time-reversal communication system.

Figure 3.12: Two methods of using time-reversal in acoustic communication. (a) Active time-reversal consists of the target emitting a signal that is recorded at an array. The time-reverse of the recorded signals at the array are then used as filters to transmit sound to the target. (b) Passive time-reversal consists of a source emitting an initial pulse, during which time the array records the response. After some time, the source transmits data and the array uses the time-reverse of the records to filter the received signals.

water environment. It was found that through the use of time-reversal it was possible to transmit signals without the need for complex channel compensation techniques. The duration for which time-reversal continued to provide sufficient filtering for it to be used in a communication system was examined by Rouseff et al. [2001] for a number of environmental conditions. In benign conditions the communication continued to operate for several seconds, however under windy conditions or when the source is drifting, the symbols became less distinguishable rather quickly.

The developments that have arisen in passive phase conjugation are as follows:

- Silva et al. [2000] presents a method known as "virtual" electronic time-reversal. This method is essentially the same as passive phase conjugation. Simulations were conducted for a single source operating at a depth of 36 m in a 40 m deep shallow water environment, transmitting to an array 2 km away having 18 elements spaced between 20 m and 36.5 m. The simulations showed that the implementation of passive TR was possible with a reduced complexity of structure compared with other filtering techniques.

- Yang [2003] discussed the Inter-Symbol-Interference (ISI) for active and passive time-reversal using simulations in conjunction with experimental work. It was found that whilst the pulse for individual channels had large side-lobes resulting in high ISI, using multiple channels with spatial diversity decreased the magnitude of the side-lobes and also phase fluctuations.

- Flynn et al. [2004] extended the PPC by combining it with an adaptive decision-directed channel-estimation technique. The technique was developed to avoid the time delay resulting from periodic channel estimation when the environment had changed and the PCC channel estimates were no longer useful. The technique involved performing PPC, and performing phase synchronisation and symbol detection after which the channel estimate is updated as the symbols are detected. An experiment was conducted in May 2000, having a range of 500 m to 5 km and the water depth varied between 10 and 120 m. The results showed outstanding performance compared to PCC.

- Yang [2004] compared passive time-reversal, and the general DFE (Decision Feedback Equaliser) in radio wave communication systems. It was found that for a small number of receivers, the passive time-reversal technique does not remove all the ISI compared with the DFE, which also resulted in a higher output SNR. However, the DFE was found to

have the following problems with numerical sensitivity with large numbers of taps, and estimation error from channel IRF variances and Doppler shifts. In some instances, the DFE did not converge with real data, as the DFE only works well with high temporal coherence. The passive time-reversal technique was found to be much more stable, so Yang [2005] coupled the passive time-reversal with a DFE. The technique developed by Yang [2005] was shown to be computationally simple, fast, and require a small number of tap co-efficients. The design was shown to be stable, with no or minimal user intervention required.

- Rouseff [2005] examined the ISI resulting from passive time-reversal with respect to a physical model of the environment. It was found that ISI was linked to three parameters: bandwidth, number of array elements, and the length of the FIR (Finite Impulse Response) matched filters. It was found that the performance had only a small dependence on array geometry, and thus receiver arrays might not necessarily be required to span the entire water column.

- Song et al. [2006b] examined a passive time-reversal technique between a source and an array whereby the source was either fixed or moving. The passive time-reversal technique was examined with and without an adaptive channel equaliser. The experiment was conducted with ranges of 4.2 km and 10 km, in 118 m deep water. When the source was moving, it was at a speed of 4 knots. The use of an adaptive equaliser with passive time-reversal always outperformed time-reversal alone, with a difference up to 13 dB and 5 dB for a moving and fixed source respectively. When an adaptive equaliser was used, two or three receivers provided reasonable performance. It was also found that the performance of time-reversal without any other equaliser saturates with no additional gain from spatial diversity for a given channel complexity.

- Song et al. [2007] demonstrated that MIMO (Multiple-Input/Multiple-Output) communication can be achieved using passive time-reversal coupled with a DFE. Experiments were conducted in 120 m deep water between two arrays moored at 4 km for one experiment, and 20 km for the second. A number of user configurations were demonstrated using a carrier frequency of 3.5 kHz and a 1 kHz bandwidth, and it was found that as many as six users could transmit over a distance of 4 km using QPSK modulation, and three users could transmit over a distance of 4 km using 16-QAM.

- Song et al. [2009] processed basin-scale data with the passive time-reversal technique. The basin-scale data was obtained from an experiment conducted in 1994 that transmitted data at 75 Hz using binary

phase shift keying over a distance of 3250 km in deep water. The information rate was 37.5 bit/s and the multi-path spanned 5 to 8 seconds. The passive time-reversal technique was able to recover the transmitted information with very few errors, demonstrating the effectiveness of time-reversal for basin-scale environments.

3.3.2 Active time-reversal in underwater acoustic communication

The earliest implementation of active time-reversal was made by Edelmann et al. [2002]. The system developed by Edelmann et al. [2002] transmitted a Binary Phase Shift Key (BPSK) signal using the time-reversal of a signal obtained from the transmission of a 2 ms, 3.5 kHz pure tone pulse and transmitting replicas of this waveform with a positive or negative scaling. Scatter plots showed that TR assists in mitigating the ISI. Active phase conjugation was found to achieve a vertical focus of less then 10 m when transmitting over 11 km in a shallow water environment having a depth that varied between 110 and 130 m. Time-reversal was compared again using a single source as a transmitter and using all the transmitters simultaneously emitting the same signal (broadside) for a number of environments. In all instances time-reversal outperformed single source and broadside transmissions.

Smith et al. [2003] examined active time-reversal in conjunction with non-coherent communication, specifically frequency shift keying (FSK) using numerical models. The temporal focusing of time-reversal overcomes the requirement for guard times that was found for FSK. The technique was found to be able to focus different messages simultaneous to different receiver locations. From simulation, it was found that:

- The size of the focus decreased in dimension as carrier frequency increased.
- The horizontal footprint was larger than the vertical footprint.
- Altering the frequency and using the same bandwidth, the temporal focusing did not change significantly.
- Varying the element spacing had a small impact on the size of the focus.
- Increasing the aperture only resulted in a small improvement in the focusing, however when the aperture was increased to $20\lambda_c$ a dramatic improvement was observed.

These outcomes were confirmed by Heinemann et al. [2003] using a small scale tank experiment.

Edelmann et al. [2005] reported on active underwater acoustic communication experiments conducted in May-June 2000. The experiments were performed over a range of 10 km in 110 m to 120 m flat shallow water and subsequently in a shallow up-slope environment. The signal operated at a carrier frequency of 3.5 kHz with a bandwidth of 500 Hz. The performance of both BPSK and QPSK (Quadrature Phase Shift Keying) modulations were investigated. Whilst the signals were able to transmit with minimal errors, Edelmann et al. [2005] considered that the major limitation to time-reversal communication was the self-generated ISI from the time-reversal process. The suggestion was made that further improvements could be obtained by using time-reversal in conjunction with a Decision Feedback Equaliser.

3.3.3 Other time-reversal investigations in underwater acoustic communication

A number of authors have investigated the application of active and passive time-reversal, or alternate implementations such as that presented by Roux et al. [2004]. An outline of the work that has been developed for general time-reversal theory follows:

- Candy et al. [2004] described how point-to-point time-reversal could be implemented in four ways: (1) Filtering at transmitter using Greens function; (2) Filtering at transmitter using probe signal; (3) Filtering at receiver using Greens function; or (4) Filtering at receiver using probe signal. It can be observed that these implementations are essentially active (1 and 2) or passive (3 and 4). Acoustic experiments were carried out in air to compare these implementations. Unfortunately the probe signal is not defined in this paper.

- Roux et al. [2004] described a technique called non-reciprocal time-reversal (NR-TR). In this technique, a pulse is emitted on each element of a source array, with a delay between each element, then the signals received at the receiver array are wirelessly transmitted back to the transmitter array for use as time-reversal filters. It was also shown that rather than transmitting the signal back wirelessly, the received signals could be used to passively determine an estimate of the original source signal using a cross-correlation at the receiver (as per passive phase conjugation).

- Candy et al. [2005] examined the spatial focusing of time-reversal using an air-acoustic experiment. The performance of the time-reversal filter was examined for various number of bits in the A/D converter. Using only 1-bit conversion degradation was observed however the signal was still reasonable and the approach could prove cost effective.

Candy et al. [2005] also extended the four methods of implementation described in Candy et al. [2004] to multiple transmitters and receivers.

- Stojanovic [2005] noted that it was often overlooked that whilst TR maximises SNR, it also increases the duration of the response. Filters were presented that maximised the SNR whilst having no ISI, or had controlled ISI that could be compensated by an equaliser. The filter structures were designed to limit the filtering to be only at the source, the receiver, or both source and receiver. Limiting the filtering at either the source or receiver was implemented to reduce the complexity, if processor power was limited. The design structures examined either multiple transmitters or multiple receivers, but never both. The filter structures developed outperformed TR which was considered severely performance limited due to ISI.

- Song et al. [2006c] presented results of underwater acoustic experiments conducted over a range of 8.6 km in 105 m deep shallow-water. The modulations used were BPSK, QPSK and 8-QAM and operated at 3.5 kHz with 1 kHz bandwidth. The implementation for transmitting the data was the same as that used by Roux et al. [2004] where the channel was recorded at the receivers, and wirelessly transmitted back to the transmitter array to transmit signals that focused on each receiver. Song et al. [2006c] found that it was possible to achieve multichannel communication using TR without any equalisation.

- Song et al. [2006a] investigated time-reversal communications with adaptive channel equalisation. Near optimal results were obtained (with respect to the optimal solution presented by Stojanovic [2005]) using this design. A conclusion suggested by Stojanovic [2005] that the receiver requires a matched filter was challenged since TR actually behaves as a matched-filter.

- Fannjiang [2006] examined MIMO time-reversal between two arrays separated by screens with pinholes, and was able to derive an upper limit for the bandwidth for this arrangement.

- Song and Kim [2007] wrote a paper that was a response to the work presented by Stojanovic [2005], whereby it was claimed that "Stojanovic [2005] *did not include important propagation physics that, if included, potentially alter some of the conclusions in* Stojanovic [2005]." The paper examined arrays capable of using the spatial diversity to compare the performance of the approaches. It is found that there are basically four different approaches: (1) TR alone; (2) TR with equalisation; (3) Equalisation with a fixed transmit array (does not use channel information); and (4) The optimal approach.

Song and Kim [2007] also critiqued the paper by Stojanovic [2005], and noted that

1. The use of four elements with a spacing of $\lambda/2$ did not provide a large enough aperture to resolve the multi-path, which is required for the TR process to compete with the other approaches.
2. Stojanovic [2005] incorrectly normalised the power for the case of a single transmitter and multiple receivers. It was stated that the transmitter power should be held constant and increasing the number of receivers actually increases the power received.

An interference pattern discussed by Stojanovic [2005] was shown to omit a frequency-dependent phase delay that results in smearing of the interference pattern.

Song and Kim [2007] found that after re-performing the simulations incorperating changes to address the points raised, it was found that the equalisers generally performed extremely similarly (including the passive time-reversal and equaliser combination). In general the TR performed better than initially portrayed by Stojanovic [2005].

3.3.4 Inverse filtering in underwater acoustic communication

The use of Tikhonov inverse filtering for acoustic communication has not received the same attention as time-reversal. The first investigation of Tikhonov inverse filtering for underwater acoustic communication was given by Cazzolato et al. [2001]. It was shown that using Tikhonov regularised inverse filters achieves better temporal focusing and slightly greater spatial focusing than TR for a simulated underwater environment. Simulations were conducted for a shallow water environment with a constant depth of 145 m. Comparisons were made between the temporal focusing for broadside, time-reversal and Tikhonov inverse filtering using two elements or sixteen elements to transmit an impulse to a probe source at a depth of 85 m located 2 km and 5 km away. Tikhonov inverse filtering was found to perform well when using both two and sixteen elements, however time-reversal was found to improve dramatically by increasing the number of transmission elements. In both cases, the focal region was slightly better for inverse filtering.

Cazzolato et al. [2001] also investigated the transmission to multiple locations (MIMO) using inverse filtering. The simulations showed that MIMO implementation of Tikhonov inverse filtering greatly reduced the cross-talk. The cross-talk ability of time-reversal was not compared, however the natural

cross-talk of independent single-channel Tikhonov inverse filtering was compared against the MIMO Tikhonov inverse filtering, where the latter proved to have much better cross-talk cancellation.

The final examination performed by Cazzolato et al. [2001] was to examine the performance of Tikhonov inverse filtering using a reduced number of bits in the A/D converters, as first suggested by Derode et al. [1999]. The simulations found that using 1-bit time-reversal introduced considerable noise into the transmission. To examine the performance of reducing the number of bits in the D/A and A/D converters, quantisation was performed for two cases: on the measured IRF used to generate the filter, and on the signal outputs from the filters. These two cases effectively simulate the reduction in the number of A/D bits for the receiver and transmitter respectively. It was found that a reduction in the number of bits during input had a more dramatic effect that on the output. Inverse filtering was found to out-perform both broadside and phase conjugation for all the combinations of bits investigated (1, 2, 4, 8 and ∞) when applied at the input or output stage, except for 1-bit output where the difference in error was minimal. Whilst all the filters were found to function with a reduced number of bits, reducing the number of bits to less than 4 bits was found to penalise the performance considerably.

The examination of Tikhonov inverse filtering for underwater acoustic communication conducted by Cazzolato et al. [2001] was limited in that the signals used in the examination consisted of single chirps rather than constant data streams. The work presented by Kim and Shin [2004] examined using a technique known as an adaptive time-reversal mirror (ATRM) that was developed by Kim et al. [2001a]. Examination of the technique shows that the design structure is closer to that of an inverse filtering technique (see Kim and Shin [2004, Eq. 10]). Kim and Shin [2004] examined the functionality of the ATRM (and thus the inverse filter) communication system using a longer data stream and found that it significantly outperformed the TR technique.

3.4 Conclusion and Gap Statement

This chapter has examined the literature concerned with the implementation of underwater acoustic communication systems. The early investigations made into underwater acoustic communication systems used the technology that had been developed for radio wave communications. A major difficulty found when implementing this technology was that the underwater environment differs from the radio wave environment since it is highly reverberant due to the large number of reflections between the sea surface and sea floor. This reverberation has resulted in the development of complicated signal processing to compensate for the reverberation. A technology known

as time-reversal has seen considerable research in recent years and provides a means of compensating for the reverberation with reduced complexity. The use of the time-reversal method also results in spatial focusing of the signal at the receiver location. Time-reversal has thus been considered as a means of increasing the transmission rate through the use of multiple transmitters and receivers. Several authors have investigated the performance of time-reversal compared with recently developed communication techniques and it has been shown that time-reversal requires additional signal processing to achieve similar performance.

Another technique know as Tikhonov inverse filtering was suggested by Cazzolato et al. [2001] as a technique that is similar to time-reversal in that it provides a reduced complexity filter design in addition to spatial focusing. Using a simulation of a single pulse transmitted in an underwater environment, it was shown that Tikhonov inverse filtering out-performed time-reversal in spatial focusing and signal quality. Whilst there has been considerable development in the use of time-reversal filtering in underwater acoustic communication systems, the Tikhonov inverse filter has not seen significant attention.

The goal of this work is to address this gap in the knowledge on the relative performance of Tikhonov inverse filters by implementing Tikhonov inverse filter design in a digital communication system and examining its performance with respect to both time-reversal, and a recently developed filter design by Stojanovic [2005]. When implementing Tikhonov inverse filtering in communication systems, a number of adjustable parameters exist that include the transducer placement, sensitivity of the transducers, parameters of the inverse filters, design structure of the inverse filter, data-rate, and carrier frequency. This research aims to investigate the influence these parameters have on the system design and its performance.

This chapter has provided a review of the literature relating to time-reversal and Tikhonov inverse filtering, along with various applications and observed features of these filter designs. This literature review has been given to provide relevant information of the recent advancements that relate to the work conducted in this thesis. This thesis expands on this previous work in Chapter 4 which examines the influences of amplifier sensitivities on Tikhonov inverse filtering; Chapter 5 which describes experiments conducted to investigate the implementation of Tikhonov inverse filtering; and Chapter 6 which presents the theory and analysis of simulation results relating to the relative performance of Tikhonov inverse filtering.

4 Influences of amplifier sensitivities on Tikhonov inverse filtering

The implementation of Tikhonov inverse filtering in acoustic communication systems is influenced by the sensitivity of the transmitter and receiver. The role of this chapter is to investigate this relationship to better understand the influence the sensitivities have on communication systems that implement Tikhonov regularised inverse filtering.

The work presented in this chapter has previously been published in the journal paper entitled *"Transducer sensitivity compensation using diagonal preconditioning for time-reversal and Tikhonov inverse filtering in acoustic systems"*, by P. Dumuid, B. Cazzolato and A. Zander, published in the Journal of Acoustical Society of America Vol. 119 (1), pp. 372-381, Jan 2006.

4.1 Introduction

The aim of the research presented in this chapter has been to investigate the implementation and performance of Tikhonov inverse filtering and similar compensation systems in conjunction with digital communication systems with specific application to shallow water acoustic environments. During the implementation of channel compensation systems in laboratory experiments it was observed that the performance of the communication system was influenced by the sensitivities of the amplifiers used for the sources (loudspeakers) and receivers (microphones).

In this chapter the influence of transducer sensitivities on the performance of these filters is examined. It is shown that the choice of transducer sensitivity has a considerable influence on the resulting filters and can negatively affect the performance of the resulting filter. To compensate for the decrease in performance, diagonal preconditioning can be implemented in the system. By using diagonal preconditioning, the loss in performance arising from unbalanced sensitivities can be reduced. An algorithm is proposed that calculates a set of diagonal matrices to precondition the channel matrix. The

algorithm is applied to a system to illustrate the technique, and the improvements of the filter performance are shown and discussed.

4.2 Theory

4.2.1 Introduction

The multi-channel filter will be discussed based on the system presented in Figure 3.10. It was shown in Section 3.2.2 that the Tikhonov regularised inverse filter design is given by

$$\mathbf{H}_{\kappa}(\omega_i) = \left(\mathbf{C}^{\mathrm{H}}(\omega_i)\mathbf{C}(\omega_i) + \kappa\mathbf{I}\right)^{-1}\mathbf{C}^{\mathrm{H}}(\omega_i). \tag{4.1}$$

where κ is the regularisation parameter, and the matrix, $\mathbf{A}(\omega)$, in Equation 3.30 has been substituted with the identity matrix, $\mathbf{I}$, since it is desired to replicate the source signals at the target receivers.

Denoting the transmitter and receiver sensitivities as $\alpha_i, i \in [1, N]$ and $\beta_j, j \in [1, M]$ respectively, the transfer matrix of the system with the sensitivities included can be expressed as

$$\begin{aligned}\mathbf{C}_{\mathrm{g}}(\omega) &= \begin{bmatrix} \beta_1 & 0 & \cdots & 0 \\ 0 & \beta_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \beta_{\mathrm{M}} \end{bmatrix} \mathbf{C} \begin{bmatrix} \alpha_1 & 0 & \cdots & 0 \\ 0 & \alpha_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \alpha_{\mathrm{N}} \end{bmatrix} \\ &= \boldsymbol{\beta}\mathbf{C}(\omega)\boldsymbol{\alpha} \end{aligned} \tag{4.2}$$

A question raised by this form is: What influence do the sensitivities have on the resulting inverse filters? In this section it will be shown that the selection of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ to achieve the smallest condition number for $\mathbf{C}_{\mathrm{g}}(\omega)$ also decreases the high regularisation needed for causality of the inverse filters that result from a poor choice of sensitivities.

In single channel systems, the coherence between the input and the output of the system is maximised by setting the sensitivity of the transmitters to the maximum value possible to reduce the noise from the electro-acoustic portion of the system, typically the largest source of noise in an acoustic system. However, with a multi-channel time-reversal or Tikhonov inverse filter design, the level of the received signal is actually determined by the filter design that is developed considering the channel and the sensitivities.

Setting the sensitivities to their maximum value for multi-channel systems does not always maximise the coherence between the input and output of the entire system consisting of the inverse filter, the sensitivities and the electro-acoustic system. It will be shown in the following sections that the algorithm

developed here will produce the optimal set of sensitivities that provide the most balanced coherence for all channels.

4.2.2 Influence of transducer sensitivities on the performance of the Tikhonov regularised inverse filter

4.2.2.1 An "equally responsive system"

The influence of diagonal preconditioning on the Tikhonov inverse filter for the conditions $\kappa = 0$, and κ tending toward infinity, shall be examined. The examination shall be performed for a system $\mathbf{C}(\omega)$ that is "equally responsive". A system shall be defined to be "equally responsive" when a signal transmitted from each input results in a similar level of excitation at each of the receivers.

When $\kappa = 0$, (i.e. no regularisation) the filter created using Equation 4.1 is found to be

$$\begin{aligned} \mathbf{H}(\omega) &= \left(\boldsymbol{\alpha}\mathbf{C}^{\mathrm{H}}(\omega)\boldsymbol{\beta}\boldsymbol{\beta}\mathbf{C}(\omega)\boldsymbol{\alpha}\right)^{-1}\boldsymbol{\alpha}\mathbf{C}^{\mathrm{H}}(\omega)\boldsymbol{\beta} \\ &= \boldsymbol{\alpha}^{-1}\left(\mathbf{C}^{\mathrm{H}}(\omega)\boldsymbol{\beta}^{2}\mathbf{C}(\omega)\right)^{-1}\mathbf{C}^{\mathrm{H}}(\omega)\boldsymbol{\beta} \end{aligned} \tag{4.3}$$

If $\boldsymbol{\beta} = \mathbf{I}$, (i.e. equal receiver sensitivities), then Equation 4.3 shows that the signal amplitude for transmitter i will be scaled by $\frac{1}{\alpha_i}$. The filter will thus create a set of signals that generates a higher signal level for the weaker transmitters. It then follows that the dynamic range will be fully utilised only for the output channel with the smallest sensitivity (assuming all the transducers have the same input dynamic range). When the matrix $\mathbf{C}$ is square (i.e. the same number of transmitters and receivers), Equation 4.3 can be reduced to

$$\mathbf{H} = \boldsymbol{\alpha}^{-1}\left(\mathbf{C}^{\mathrm{H}}(\omega)\mathbf{C}(\omega)\right)^{-1}\mathbf{C}^{\mathrm{H}}(\omega)\boldsymbol{\beta}^{-1} \tag{4.4}$$

showing that a similar attenuation is applied to the input signal according to the choice of receiver sensitivities, $\boldsymbol{\beta}$.

When regularisation is included, it can be noted from consideration of Equation 4.1 that as κ is increased, $\left(\mathbf{C}_{\mathrm{g}}^{\mathrm{H}}\mathbf{C}_{\mathrm{g}} + \kappa\mathbf{I}\right)^{-1}$ tends toward $\frac{1}{\kappa}\mathbf{I}$, and as a result, the resulting filter approaches

$$\mathbf{H}(\omega) = \frac{1}{\kappa}\mathbf{C}_{\mathrm{g}}^{\mathrm{H}}(\omega) \tag{4.5}$$

$$= \frac{1}{\kappa}\boldsymbol{\alpha}\mathbf{C}^{\mathrm{H}}(\omega)\boldsymbol{\beta} \tag{4.6}$$

which can be observed to be a scaled version of the frequency domain representation of the multi-channel time-reversal filter [Kuperman et al., 1998,

Jackson and Dowling, 1991, Prada et al., 1996],

$$\mathbf{H}(\omega) \;=\; \mathbf{C}^{\mathrm{H}}(\omega). \tag{4.7}$$

The effect of the sensitivities on this filter design is that the signal to transducer i is scaled by α_i, and the signal transmitted to receiver j is scaled by β_j. It then follows that the dynamic range will only be fully utilised for the output channel with the largest sensitivity.

It has been shown that at the two extremities of $\kappa = 0$ and $\kappa \to \infty$, (denoted hereafter as inverse filtering (IF) and time-reversal filtering (TRF)), the full dynamic range of the transducer will only be effectively used if the transducer sensitivities are equal for an "equally responsive system". It can then be noted that if the system is not "equally responsive" (i.e. a source or receiver positioned close to a pressure node), it would be desirable to find an alternative set of sensitivities that would transform the total system into an "equally responsive system".

4.2.2.2 Influence of transducer sensitivities on the total system

With reference to Figure 3.10, the total system transfer function, being the combination of the filter and the system, is given by

$$\mathbf{T}(\omega) \;=\; \mathbf{C}(\omega)\mathbf{H}(\omega). \tag{4.8}$$

The influence that the transducer sensitivities have on the total system for IF and TRF can be observed by inserting Equations 4.3 and 4.6 into Equation 4.8. The system transfer functions for IF and TRF are given by

$$\begin{aligned} \mathbf{T}_{\mathrm{IF}} &= \boldsymbol{\beta}\mathbf{C}\boldsymbol{\alpha}\mathbf{H}_{\mathrm{IF}} \\ &= \boldsymbol{\beta}\mathbf{C}\boldsymbol{\alpha}\left(\left(\boldsymbol{\alpha}\mathbf{C}^{\mathrm{H}}\boldsymbol{\beta}\boldsymbol{\beta}\mathbf{C}\boldsymbol{\alpha}\right)^{-1}\boldsymbol{\alpha}\mathbf{C}^{\mathrm{H}}\boldsymbol{\beta}\right) \\ &= \mathbf{I} \end{aligned} \tag{4.9}$$

and

$$\begin{aligned} \mathbf{T}_{\mathrm{TRF}} &= \boldsymbol{\beta}\mathbf{C}\boldsymbol{\alpha}\mathbf{H}_{\mathrm{TR}} \\ &= \boldsymbol{\beta}\mathbf{C}\boldsymbol{\alpha}\left(\boldsymbol{\alpha}\mathbf{C}^{\mathrm{H}}\boldsymbol{\beta}\right) \\ &= \boldsymbol{\beta}\left(\sum_{i=1}^{\mathrm{N}}\alpha_i^2\begin{bmatrix} c_{1i} \\ c_{2i} \\ \vdots \end{bmatrix}\begin{bmatrix} c_{1i}^* & c_{2i}^* & \cdots \end{bmatrix}\right)\boldsymbol{\beta} \end{aligned} \tag{4.10}$$

respectively, and the matrix

$$\begin{bmatrix} c_{1i} \\ c_{2i} \\ \vdots \end{bmatrix}\begin{bmatrix} c_{1i}^* & c_{2i}^* & \cdots \end{bmatrix} \tag{4.11}$$

is the transfer matrix due to the ith transmitter. It is thus observed that the variation of the transducer sensitivities has no influence on the total response for an IF but considerable influence on the TRF.

Considering that $\mathbf{C}\mathbf{C}^{\mathrm{H}}$ is diagonally dominant [Tanter et al., 2000], Equation 4.10 shows that the transducer sensitivities $\boldsymbol{\beta}$ result in the signal at the jth receiver being scaled by β_j^2, and the sensitivities $\boldsymbol{\alpha}$ result in the scaling of the ith transfer matrix by α_i^2. Since in practise the Tikhonov inverse filter has a non-zero regularisation parameter, it is considered reasonable to assume that the transmission channels would also be unequally scaled.

4.2.2.3 Examination of the transfer matrix singular values

In this section, the influence of the transducer sensitivities on the Tikhonov IF will be examined according to the singular value decomposition (SVD) of the system matrix, given by

$$\begin{aligned} \mathbf{C}(\omega) &= \mathbf{U}(\omega)\boldsymbol{\Sigma}(\omega)\mathbf{V}^{\mathrm{H}}(\omega) \\ &= \sum_{i=1}^{N} \sigma_i \mathbf{u}_i(\omega)\mathbf{v}_i^{\mathrm{H}}(\omega) \end{aligned} \tag{4.12}$$

where $\mathbf{U}(\omega)$ and $\mathbf{V}(\omega)$ are unitary matrices, $\boldsymbol{\Sigma}(\omega)$ a diagonal matrix of singular values, $\sigma_i, i \in [1, N]$, and $\mathbf{u}_i(\omega)$ and $\mathbf{v}_i(\omega)$ are the corresponding basis vectors within the unitary matrices. The inverse filter with no regularisation can then be expressed as

$$\begin{aligned} \mathbf{H}_{\mathrm{IF}}(\omega) &= \mathbf{V}(\omega)\boldsymbol{\Sigma}^{-1}(\omega)\mathbf{U}^{\mathrm{H}}(\omega) \\ &= \sum_{i=1}^{N} \frac{\mathbf{v}_i(\omega)\mathbf{u}_i^{\mathrm{H}}(\omega)}{\sigma_i} \end{aligned} \tag{4.13}$$

and the addition of the regularisation results in the filter

$$\begin{aligned} \mathbf{H}_{\mathrm{TIF}}(\omega) &= \mathbf{V}(\omega)\boldsymbol{\Sigma}_{\mathrm{TIF}}(\omega)\mathbf{U}^{\mathrm{H}}(\omega) \\ &= \sum_{i=1}^{N} \left(\frac{\sigma_i^2}{\sigma_i^2 + \kappa} \right) \frac{\mathbf{v}_i(\omega)\mathbf{u}_i^{\mathrm{H}}(\omega)}{\sigma_i} \end{aligned} \tag{4.14}$$

where the subscript TIF denotes Tikhonov inverse filter. In subsequent equations the frequency dependence, (ω) is implied, but not shown. In Equation 4.14 it can be seen that the magnitude of κ compared to σ_i^2 (the singular values of $\mathbf{C}(\omega)\mathbf{C}^{\mathrm{H}}(\omega)$) determines the effectiveness of the "basis vector coupling" between $\mathbf{u}_i$ and $\mathbf{v}_i$ [Tanter et al., 2001]. "Basis vector coupling" is physically described as follows: $\mathbf{u}_i$ is considered similar to a mode shape that, when excited, results in an excitation of the receivers with a phase and amplitude, $\mathbf{v}_i$, scaled according to the coupling factor of σ_i.

When the sensitivities are included, the filter becomes

$$\mathbf{C}_{\mathrm{g}} = \boldsymbol{\beta}\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{\mathrm{H}}\boldsymbol{\alpha} \quad (4.15)$$
$$= \mathbf{U}_{\mathrm{g}}\boldsymbol{\Sigma}_{\mathrm{g}}\mathbf{V}_{\mathrm{g}}^{\mathrm{H}} \quad (4.16)$$

where $\mathbf{U}_{\mathrm{g}}$,$\mathbf{V}_{\mathrm{g}}$ and $\boldsymbol{\Sigma}_{\mathrm{g}}$ are the unitary and singular matrices of the new system. The basis vector coupling matrices, $\mathbf{u}_i\sigma_i\mathbf{v}_i^{\mathrm{H}}$, have been converted to $\boldsymbol{\beta}\mathbf{u}_i\sigma_i\mathbf{v}_i^{\mathrm{H}}\boldsymbol{\alpha}$. Since the set of vectors, $\boldsymbol{\beta}\mathbf{u}_i, i \in [1, \mathrm{M}]$ and $\boldsymbol{\alpha}\mathbf{v}_i, i \in [1, \mathrm{N}]$ (being the transformation of the original basis vectors), cannot be simply scaled to form another orthonormal set, it can be concluded that there is no trivial solution to relate the singular values of $\mathbf{C}$ to those of $\mathbf{C}_{\mathrm{g}}$.

In this work, the goal is to determine a new set of sensitivities that reduce the regularisation that results from a poor choice of sensitivities. Given a fixed regularisation parameter, κ, Equation 4.14 shows that to reduce the effect of the regularisation on the singular values, sensitivities should be chosen that result in the largest singular values possible. This strategy by itself is unrealistic because the problem is unconstrained since $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ can be chosen to scale the singular values by any desired amount, x, by using a set of scaling matrices,

$$\boldsymbol{\alpha} = x\mathbf{I}, \qquad \boldsymbol{\beta} = \mathbf{I}, \quad (4.17)$$

It can be further shown using Equation 4.14, that the change in regularisation that results from scaling the sensitivities by x can equivalently be achieved by selecting a different regularisation parameter, $\kappa' = x^2\kappa$. Thus, the objective of adjusting the sensitivities should not be to scale the singular values, but to minimise the condition number, being the ratio of the largest and smallest singular values.

4.2.3 Calculation of desirable transducer sensitivities

In this section it will be assumed that it is possible to alter the sensitivities of the transducers in the system by altering the sensitivity of the amplifiers. In Section 4.2.2, two sets of ideal transducer sensitives were proposed that: (1) Achieve an "equally responsive system", and (2) Reduce the condition number of the matrix.

4.2.3.1 Sensitivities for an "equally responsive system"

In order to have an "equally responsive system", the transducer sensitivities are chosen such that every input signal to the system excites the outputs of the system with the same magnitude. This can be expressed as

$$|\boldsymbol{\beta}\mathbf{C}\boldsymbol{\alpha}\mathbf{e}_1|_2 = |\boldsymbol{\beta}\mathbf{C}\boldsymbol{\alpha}\mathbf{e}_2|_2 = \cdots = |\boldsymbol{\beta}\mathbf{C}\boldsymbol{\alpha}\mathbf{e}_{\mathrm{N}}|_2 \quad (4.18)$$

where $|\cdot|_2$ is the norm-2 (or Euclidean length) of a vector, and the vectors $\mathbf{e}_1, \ldots, \mathbf{e}_\mathrm{N}$ are the standard basis vectors for $\mathbb{R}_\mathrm{N}$. This condition can be achieved by setting

$$\begin{aligned} \beta_i &= 1 \\ \alpha_j^2 &= \frac{1}{\sum_{i=1}^{\mathrm{M}} |c_{ij}|^2} \end{aligned} \tag{4.19}$$

By using this scaling, the resulting filters will equalise the signals transmitted, but not the signals received. In order to achieve equal signal levels at the receivers, a further condition can be imposed: for a simultaneous unit input on all the channels, the energy at each output is to be equal. To achieve this, a set of diagonal matrices, $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, are chosen such that

$$r_1^2 = r_2^2 = \cdots = r_\mathrm{M}^2 \tag{4.20}$$

where

$$\begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_\mathrm{M} \end{bmatrix} = \boldsymbol{\beta}\mathbf{C}\boldsymbol{\alpha} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \tag{4.21}$$

A solution that achieves this is

$$\begin{aligned} \alpha_j &= 1 \\ \beta_i^2 &= \frac{1}{\sum_{j=1}^{\mathrm{N}} |c_{ij}|^2} \end{aligned} \tag{4.22}$$

If the conditions in Equations 4.19 and 4.22 are met, the system responds equally, and thus Tikhonov inverse filtering can be found to effectively use the full dynamic range of all the transducers within the system.

In order to obtain a perfect "equally responsive system", both Equations 4.19 and 4.22 would need to be simultaneously met. However, it is not always possible to meet both these conditions, and an algorithm is presented in the following section that attempts to provide a close approximation.

4.2.3.2 Sensitivities to reduce the condition number of the system

In Section 4.2.2.3 it was shown that a suitable choice of diagonal matrices was the set that minimised the condition numbers of the system. Van der Sluis [1969] discussed that minimisation of the condition number could not be expected to be easily achieved, however it was shown [Van der Sluis,

1969, Theorem 3.5] that the condition number of the matrix $\boldsymbol{\alpha}\mathbf{C}$ was upper bounded to be a factor of $\sqrt{m}$ from the minimum when all the rows have equal 2-norms, and the condition number of $\mathbf{C}\boldsymbol{\beta}$ was upper bounded to be a factor of $\sqrt{n}$ from the minimum when all the columns have equal 2-norms for an $m \times n$ $\mathbf{C}$ matrix. From Equations 4.19 and 4.22, the diagonal matrices that best use the dynamic range of the transducers also results in a matrix of equal 2-norm of both the rows and columns. Thus the design techniques presented in Sections 4.2.2.2 and 4.2.2.3 have the same solution, being that of diagonal matrices that result in $\mathbf{C}_{\mathrm{g}}$ having rows and columns of equal 2-norm.

Finding a set of diagonal matrices that achieve equal 2-norms of both the columns and rows simultaneously is a non-trivial problem. In order to approximate such a condition, Ruiz [2001] presented an algorithm that applies Equations 4.19 and 4.22 iteratively and converges to a set of diagonal matrices having equal 2-norms of both rows and columns of the combined matrices. The algorithm by Ruiz [2001] is considered as a suitable means to calculate an optimal set of diagonal matrices and is presented as follows:

Algorithm 1

$$\hat{\mathbf{C}}^{(0)} = \mathbf{C}, \quad \boldsymbol{\beta}^{(0)} = \mathbf{I}, \quad \boldsymbol{\alpha}^{(0)} = \mathbf{I}$$

for $k = 0, 1, 2, \ldots,$ until convergence do:

$$\begin{aligned} \mathbf{D}_{\mathrm{R}} &= \operatorname{diag}\left(\sqrt{\left|\mathbf{r}_i^{(k)}\right|_2}\right)_{i=1,\ldots,m}, \text{ and} \\ \mathbf{D}_{\mathrm{C}} &= \operatorname{diag}\left(\sqrt{\left|\mathbf{c}_j^{(k)}\right|_2}\right)_{j=1,\ldots,n} \end{aligned}$$

$$\hat{\mathbf{C}}^{(k+1)} = \mathbf{D}_{\mathrm{R}}^{-1}\hat{\mathbf{C}}^{(k)}\mathbf{D}_{\mathrm{C}}^{-1}$$

$$\boldsymbol{\beta}^{(k+1)} = \boldsymbol{\beta}^{(k)}\mathbf{D}_{\mathrm{R}}^{-1} \quad \text{, and} \quad \boldsymbol{\alpha}^{(k+1)} = \boldsymbol{\alpha}^{(k)}\mathbf{D}_{\mathrm{C}}^{-1}$$

where $\mathbf{r}_i^{(k)}$ and $\mathbf{c}_j^{(k)}$ are the ith row and jth column of the matrix $\hat{\mathbf{C}}^{(k)}$ respectively. For the experimental results given in Section 4.3, it was found that adequate convergence of the algorithm was reached after 20 iterations, after which further iterations had little influence on the magnitudes of the values in the matrices.

4.2.4 Implementations of preconditioning in digital systems

So far the implementation of sensitivity compensation has only been discussed with respect to scaling within the analog domain. In this section the concept of scaling the signal within the digital domain will be presented. Figure 4.1 shows a number of variations of how preconditioning can be performed in the analog and digital domain.

To develop a filter for use in the digital domain, it is observed that the filter, $\mathbf{H}_{\mathrm{g}}$, is designed such that

$$[\boldsymbol{\beta}\mathbf{C}\boldsymbol{\alpha}]\,\mathbf{H}_{\mathrm{g}} \;\simeq\; \mathbf{I} \tag{4.23}$$

where the square brackets have been included to denote the analog domain. It then follows that

$$\begin{aligned} \boldsymbol{\beta}^{-1}\boldsymbol{\beta}\mathbf{C}\boldsymbol{\alpha}\mathbf{H}_{\mathrm{g}} &\simeq \boldsymbol{\beta}^{-1} \\ [\mathbf{C}]\,\boldsymbol{\alpha}\mathbf{H}_{\mathrm{g}}\boldsymbol{\beta} &\simeq \mathbf{I} \end{aligned} \tag{4.24}$$

Thus an inverse filter for use in the digital domain is given by

$$\begin{aligned} \mathbf{H}_{\mathrm{digital}} &= \boldsymbol{\alpha}\mathbf{H}_{\mathrm{g}}\boldsymbol{\beta} \\ &= \boldsymbol{\alpha}\left((\boldsymbol{\beta}\mathbf{C}\boldsymbol{\alpha})^{\mathrm{H}}(\boldsymbol{\beta}\mathbf{C}\boldsymbol{\alpha}) + \kappa\mathbf{I}\right)^{-1}(\boldsymbol{\beta}\mathbf{C}\boldsymbol{\alpha})^{\mathrm{H}}\boldsymbol{\beta} \\ &= \left(\mathbf{C}^{\mathrm{H}}\boldsymbol{\beta}^{2}\mathbf{C} + \kappa\boldsymbol{\alpha}^{-2}\right)^{-1}\mathbf{C}^{\mathrm{H}}\boldsymbol{\beta}^{2} \end{aligned} \tag{4.25}$$

The digital and analog implementations are shown in Figure 4.1b and 4.1c, where, $\mathbb{H}\{\,\}$, is the Tikhonov regularised inverse filter operator defined as

$$\mathbb{H}\{\mathbf{X}\} = \left(\mathbf{X}^{\mathrm{H}}\mathbf{X} + \kappa\mathbf{I}\right)^{-1}\mathbf{X}^{\mathrm{H}}. \tag{4.26}$$

When applying diagonal preconditioning in the analog domain, $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are chosen to transform the system $\mathbf{C}_{\mathrm{g}}(z)$ into an "equally responsive system" such that the signals at the input and output of the system have relatively equal amplitudes. However, if the scaling is performed within the digital domain, the amplitude of the signals at the D/A and A/D converters are

$$\mathbf{s}_{\mathrm{D/A}}(z) \;=\; \boldsymbol{\alpha}\mathbf{v}(z) \tag{4.27}$$

and

$$\mathbf{s}_{\mathrm{A/D}}(z) \;=\; \boldsymbol{\beta}^{-1}\mathbf{w}(z) \tag{4.28}$$

respectively, showing that the filter does not make effective use of the D/A and A/D converters. Thus the only benefit to using diagonal preconditioning in the digital domain is to reduce the unequal regularisation on the singular values which results from a poor choice of sensitivities.

Note that the scaling arrangement (d) shown in Figure 4.2 will be addressed in the next section.

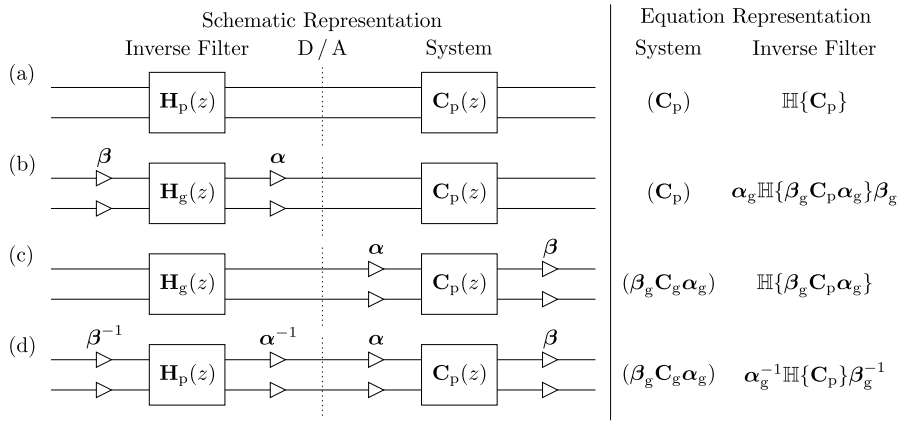


Figure 4.1: Diagonal preconditioning systems: (a) no diagonal preconditioning; (b) digital preconditioning; (c) analog preconditioning; and (d) scaled version of the Tikhonov inverse filter.

4.3 An example analysis

In this section, a simulation will be used to demonstrate the concept of diagonal preconditioning. The simulation repeats the simulation performed by Kirkeby et al. [1998], however the data has been altered to emulate a system with incorrect amplifier sensitivities. The preconditioning algorithm is then used to correct the amplifier sensitivities. The simulation consists of using Tikhonov regularised inverse filtering with four speakers to generate a set of desired signals at four points surrounding a dummy head. For a detailed overview of the physical configuration, see Kirkeby et al. [1998]. The simulation utilises transfer functions created by Gardner and Martin [1994] which are freely available for download from the MIT Media Laboratory website (World Wide Web Address: `http://sound.media.mit.edu/KEMAR.html`). The impulse responses that describe the system are shown in Figure 4.2. It should be observed that due to symmetry in the experiment, the impulse response matrix can be approximately written in the form

$$\mathbf{C}(z) = \begin{bmatrix} c_1(n) & c_2(n) & c_3(n) & c_4(n) \\ c_2(n) & c_1(n) & c_4(n) & c_3(n) \\ c_5(n) & c_6(n) & c_7(n) & c_8(n) \\ c_6(n) & c_5(n) & c_8(n) & c_7(n) \end{bmatrix}. \tag{4.29}$$

It can be observed from Figure 4.2 that the energies of $c_1(n), c_3(n), c_5(n)$, and $c_7(n)$ are relatively equal, and similarly the energies of $c_2(n), c_4(n), c_6(n)$, and $c_8(n)$ are relatively equal. It can thus be concluded that the norm-2 of the rows and columns of this matrix are likely to be fairly similar, and thus the system is already "equally responsive". To examine the influence of diagonal preconditioning, a set of sensitivities will be used to cause the system to be

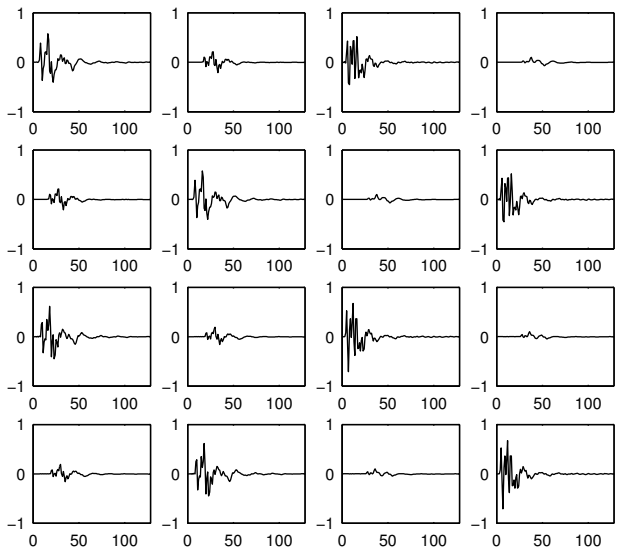


Figure 4.2: The impulse responses $c_{rs}(n)$ of the system (replica of Kirkeby et al. [1998, Fig. 3]), showing the response amplitudes versus sample, n. In this figure, the sub-figure at row i, column j corresponds to the IRF of the channel between transmitter j and receiver i.

poorly scaled and a set of sensitivities is calculated using Algorithm 1 to compensate for the poor scaling.

The set of sensitivities arbitrarily chosen to create a system with poor scaling is given by

$$\begin{aligned} \boldsymbol{\alpha}_{\mathrm{p}} &= \begin{bmatrix} 1.00 & 0 & 0 & 0 \\ 0 & 0.50 & 0 & 0 \\ 0 & 0 & 1.00 & 0 \\ 0 & 0 & 0 & 1.00 \end{bmatrix} \\ \boldsymbol{\beta}_{\mathrm{p}} &= \begin{bmatrix} 0.25 & 0 & 0 & 0 \\ 0 & 1.00 & 0 & 0 \\ 0 & 0 & 1.00 & 0 \\ 0 & 0 & 0 & 1.00 \end{bmatrix} \end{aligned} \tag{4.30}$$

with the resulting IRFs shown in Figure 4.3a. This system will be denoted as $\mathbf{C}_{\mathrm{p}}$, where the subscript p denotes *poorly scaled*. The scaling physically corresponds to transmitter 2 having half the sensitivity of the other transmitters and receiver 1 having a sensitivity a quarter that of the other elements. A set of compensating sensitivities were then calculated by applying Algorithm 1

Table 4.1: Energy within the rows and columns of the transfer matrices

System	Rows	Columns
Poorly scaled system ($\mathbf{C}_\mathrm{p}$)	$\left[\,0.70\ 0.44\ 0.77\ 1.00\,\right]^\mathrm{T}$	$\left[\,0.24\ 0.81\ 1.00\ 0.86\,\right]$
Compensated $\left(\boldsymbol{\beta}_\mathrm{g}\mathbf{C}_\mathrm{p}\boldsymbol{\alpha}_\mathrm{g}\right)$	$\left[\,1.00\ 0.98\ 1.00\ 0.98\,\right]^\mathrm{T}$	$\left[\,0.98\ 1.00\ 0.98\ 1.00\,\right]$

to the system root-mean square matrix,

$$\mathbf{E} = \begin{bmatrix} \sqrt{\sum_n c_{11}^2(n)} & \sqrt{\sum_n c_{12}^2(n)} & \sqrt{\sum_n c_{13}^2(n)} & \sqrt{\sum_n c_{14}^2(n)} \\ \sqrt{\sum_n c_{21}^2(n)} & \sqrt{\sum_n c_{22}^2(n)} & \sqrt{\sum_n c_{23}^2(n)} & \sqrt{\sum_n c_{24}^2(n)} \\ \sqrt{\sum_n c_{31}^2(n)} & \sqrt{\sum_n c_{32}^2(n)} & \sqrt{\sum_n c_{33}^2(n)} & \sqrt{\sum_n c_{34}^2(n)} \\ \sqrt{\sum_n c_{41}^2(n)} & \sqrt{\sum_n c_{42}^2(n)} & \sqrt{\sum_n c_{43}^2(n)} & \sqrt{\sum_n c_{44}^2(n)} \end{bmatrix}. \tag{4.31}$$

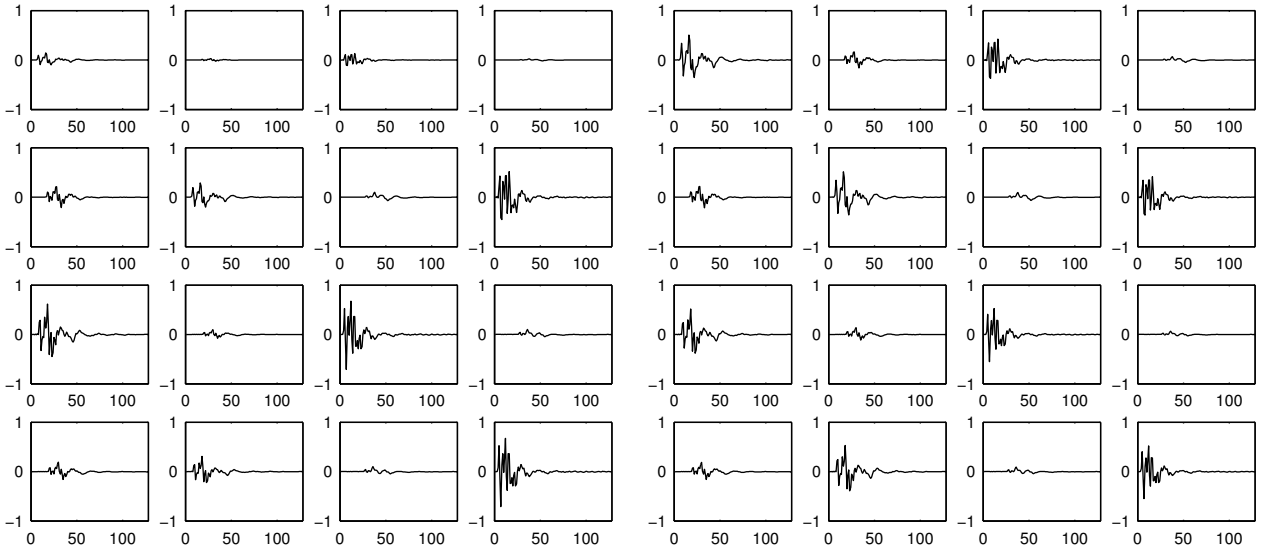
The resulting compensation sensitivities are

$$\begin{aligned} \boldsymbol{\alpha}_\mathrm{g} &= \begin{bmatrix} 1.06 & 0 & 0 & 0 \\ 0 & 1.85 & 0 & 0 \\ 0 & 0 & 0.99 & 0 \\ 0 & 0 & 0 & 0.83 \end{bmatrix} \\ \boldsymbol{\beta}_\mathrm{g} &= \begin{bmatrix} 3.30 & 0 & 0 & 0 \\ 0 & 0.96 & 0 & 0 \\ 0 & 0 & 0.79 & 0 \\ 0 & 0 & 0 & 0.93 \end{bmatrix}. \end{aligned} \tag{4.32}$$

Figure 4.3b shows the IRFs of the system after sensitivity compensation has been applied (i.e. the application of $\boldsymbol{\alpha}_\mathrm{p}\boldsymbol{\alpha}_\mathrm{g}$ and $\boldsymbol{\beta}_\mathrm{p}\boldsymbol{\beta}_\mathrm{g}$ to the initial system.) Table 4.1 shows the energy within the rows and columns of both systems, normalised such that the largest energy level is unity. As the energy within each row and each column for the sensitivity compensated system are of similar magnitude (in contrast to that of the poorly scaled system), the algorithm is thus observed to work as desired.

The singular values of the two systems as a function of frequency are shown in Figure 4.4. It can be observed in this figure that when the system is poorly scaled, the spread of the singular values is much larger and therefore the condition number is higher than that obtained when compensation sensitivities are used.

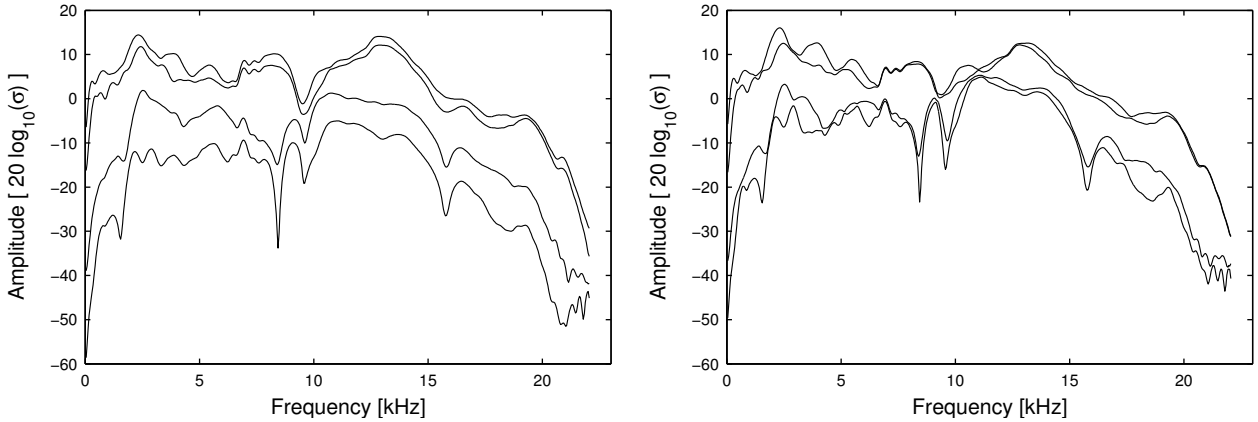
Figure 4.5 shows the sensitivities, $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, that would result in the optimal scaling for each particular frequency. It can be observed that with sensitivity compensation, the spread of these curves is reduced. If the system is to be used for band-limited operation, then in practise a choice of sensitivities would be found by averaging $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ over the desired bandwidth of operation.



(a) Poorly scaled system, $\mathbf{C}_\mathrm{p}$

(b) Poorly scaled system after application of sensitivity compensation, $\boldsymbol{\beta}_\mathrm{g}\mathbf{C}_\mathrm{p}\boldsymbol{\alpha}_\mathrm{g}$

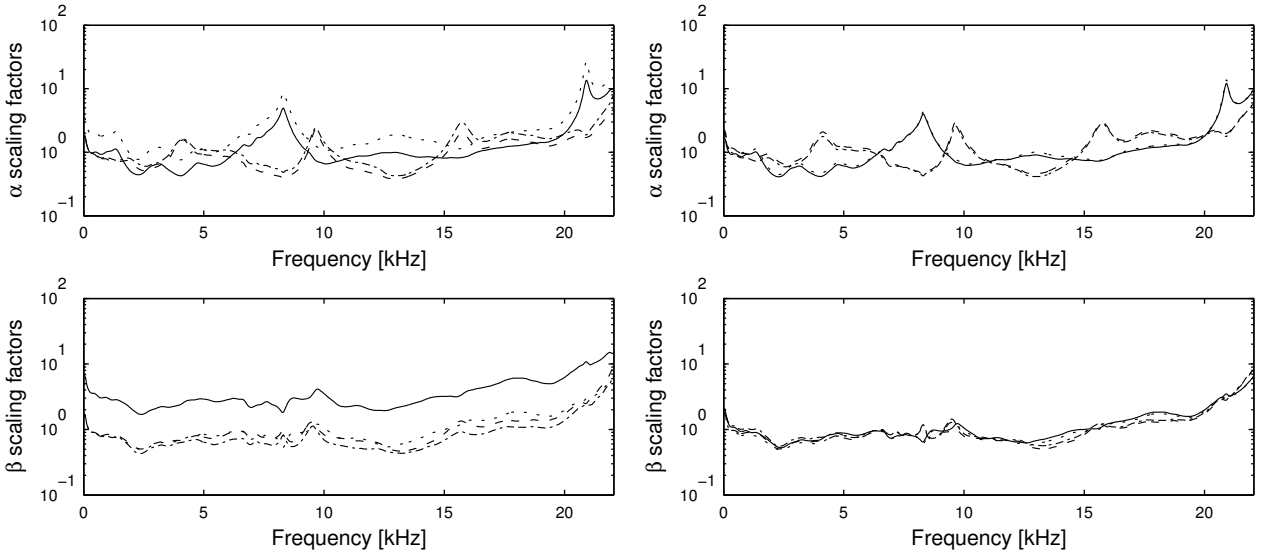
Figure 4.3: The impulse responses $c(n)$. In these figures, the subplot at row i, column j corresponds to the IRF of the channel between transmitter j and receiver i.



(a) Poorly scaled system, $\mathbf{C}_\mathrm{p}$

(b) Poorly scaled system after application of sensitivity compensation, $\boldsymbol{\beta}_\mathrm{g}\mathbf{C}_\mathrm{p}\boldsymbol{\alpha}_\mathrm{g}$

Figure 4.4: The singular values of $\mathbf{C}(\omega)$.



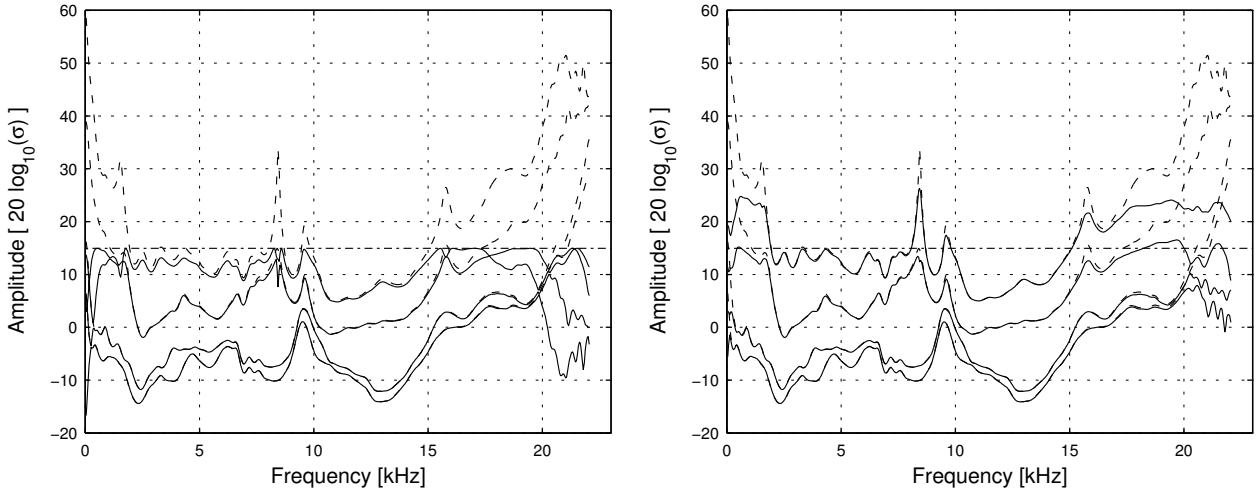
(a) Poorly scaled system, $\mathbf{C}_\mathrm{p}$

(b) Poorly scaled system after application of sensitivity compensation, $\boldsymbol{\beta}_\mathrm{g}\mathbf{C}_\mathrm{p}\boldsymbol{\alpha}_\mathrm{g}$

Figure 4.5: Optimal values of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ with respect to frequency, calculated using the preconditioning algorithm. ——— x_1, · · · · · · x_2, -·-·-·- x_3, - - - - x_4 where $x = \alpha$ and β respectively.

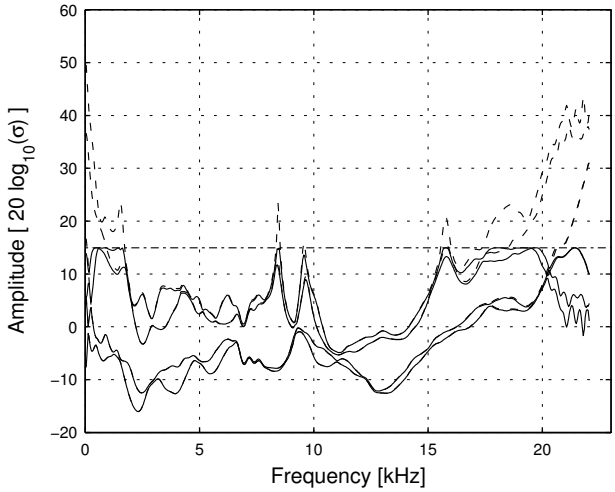
In Section 4.2.4 it was shown that diagonal preconditioning could be performed in either the digital or analog domain. To understand the influence of diagonal preconditioning, the various implementations shown in Figure 4.1 were examined. In order to compare the performance of the filter with and without diagonal preconditioning, the systems that the filters are compensating for should be identical. When diagonal preconditioning is implemented in the digital domain (Figure 4.1b), the system being compensated is the same as that without preconditioning (Figure 4.1a). However, when diagonal preconditioning is implemented in the analog domain (Figure 4.1c), the system being compensated is different. In order to have a benchmark against which the performance of the analog implementation can be compared, a new filter is introduced, being the Tikhonov inverse filter formed from the system with no preconditioning scaled for a system with poor sensitivities using the same method and assumptions used to obtain Equation 4.25. The schematic of this configuration is shown in Figure 4.1d.

The singular values curves calculated for each of the filters presented in Figure 4.1 are shown in Figure 4.6. These curves represent the "basis vector coupling" discussed in Section 4.2.2.3. Figures 4.6a and 4.6b show the singular values of the inverse filters designed to compensate for the poorly scaled system, whilst Figures 4.6c and 4.6d show the singular values of the

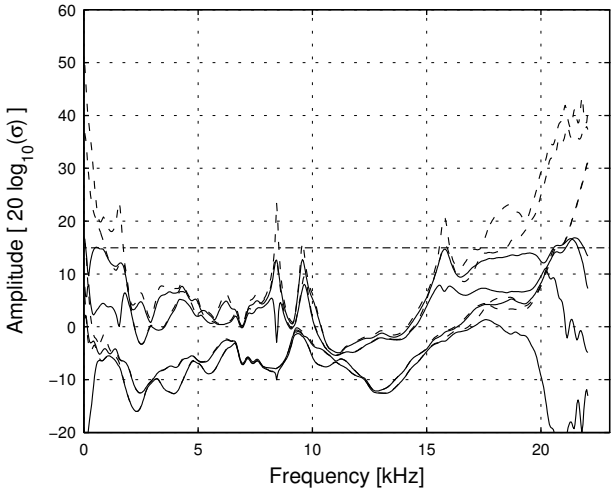


(a) No preconditioning, $\mathbb{H}\{\mathbf{C}_\mathrm{p}\}$

(b) Digital preconditioning, $\boldsymbol{\alpha}_\mathrm{g}\mathbb{H}\left\{\boldsymbol{\beta}_\mathrm{g}\mathbf{C}_\mathrm{p}\boldsymbol{\alpha}_\mathrm{g}\right\}\boldsymbol{\beta}_\mathrm{g}$



(c) Analog preconditioning, $\mathbb{H}\left\{\boldsymbol{\beta}_\mathrm{g}\mathbf{C}_\mathrm{p}\boldsymbol{\alpha}_\mathrm{g}\right\}$



(d) No preconditioning, scaled, $\boldsymbol{\alpha}_\mathrm{g}^{-1}\mathbb{H}\{\mathbf{C}_\mathrm{p}\}\boldsymbol{\beta}_\mathrm{g}^{-1}$

Figure 4.6: The singular values of the $\mathbf{H}_\mathrm{TIF}(\omega_n)$ for $\kappa = 0.008$, ——— with regularisation, - - - - without regularisation, -·-·-·- singular value limit, $\frac{1}{2\sqrt{\kappa}}$.

inverse filters designed to compensate for a system incorporating sensitivity compensation.

Figures 4.6a and 4.6c show that the filters do not have any singular values that exceed 15 dB. This limit can be explained with reference to Equations 4.13 and 4.14 where regularisation changes the singular value of the inverse filter from

$$\sigma_\mathrm{IF} = \frac{1}{\sigma_\mathrm{C}} \tag{4.33}$$

to

$$\sigma_{\mathrm{TIF}} = \frac{\sigma_{\mathrm{C}}^2}{\left(\sigma_{\mathrm{C}}^2 + \kappa\right)\sigma_{\mathrm{C}}} \tag{4.34}$$

where σ_{C}, σ_{IF} and σ_{TIF} and are the singular values of the channel, inverse filter and the Tikhonov inverse filter respectively. A plot of these functions is given in Figure 4.7 for $\kappa = 0.008$. Equation 4.34 is observed to limit the maximum possible singular value of the inverse filter. The maximum possible singular value in the Tikhonov regularised inverse filter for a given regularisation, κ, is the value of σ_{TIF} when $\frac{\partial}{\partial \sigma_{\mathbf{C}}}\sigma_{\mathrm{TIF}} = 0$ and can be derived as follows: Taking the partial derivative of the singular values gives

$$\begin{aligned} \frac{\partial}{\partial \sigma_{\mathrm{C}}}\sigma_{\mathrm{TIF}} &= \frac{1}{\left(\sigma_{\mathrm{C}}^2 + \kappa\right)} - \frac{2\sigma_{\mathrm{C}}^2}{\left(\sigma_{\mathrm{C}}^2 + \kappa\right)^2} \\ &= \frac{\left(\sigma_{\mathrm{C}}^2 + \kappa\right) - 2\sigma_{\mathrm{C}}^2}{\left(\sigma_{\mathrm{C}}^2 + \kappa\right)^2}, \end{aligned} \tag{4.35}$$

setting $\frac{\partial}{\partial \sigma_{\mathrm{C}}}\sigma_{\mathrm{TIF}} = 0$

$$\begin{aligned} \left(\sigma_{\mathrm{C}}^2 + \kappa\right) - 2\sigma_{\mathrm{C}}^2 &= 0 \\ \sigma_{\mathrm{C}} &= \sqrt{\kappa} \end{aligned} \tag{4.36}$$

and inserting σ_{C} into Equation 4.34, we obtain

$$\sigma_{\mathrm{TIF}} = \frac{1}{2\sqrt{\kappa}}. \tag{4.37}$$

Thus the singular value for the regularisation, $\kappa = 0.008$ is limited to 15 dB which can be confirmed in Figure 4.7. When $\sigma_{\mathrm{C}} > \sqrt{\kappa}$, the singular values are reflected about $\frac{1}{2\sqrt{\kappa}}$. This limit is shown in Figure 4.6 as a dotted horizontal line. Comparing Figures 4.6a and 4.6b, the singular values when using sensitivity compensation are no longer limited at 15 dB, but rather a regularisation is evident that takes into account the poor choice of sensitivities in the system.

Figures 4.6c and 4.6d show the singular value curves of the inverse filters designed to compensate for a system incorporating sensitivity compensation. Figure 4.6c shows the singular values for the inverse filter that was designed using the channel response incorporating sensitivity compensation, whilst Figure 4.6d shows the singular values for the inverse filter developed using the poorly scaled system and scaled to suit the system that has incorporated sensitivity compensation. The inverse filter design when the system had poor scaling has been regularised considerably (Figure 4.6d) compared to the filter from the system having sensitivity compensation (Figure 4.6c). The regularisation of the inverse filter designed when the system had poor scaling is particularly visible on the lowest curve above 15 kHz.

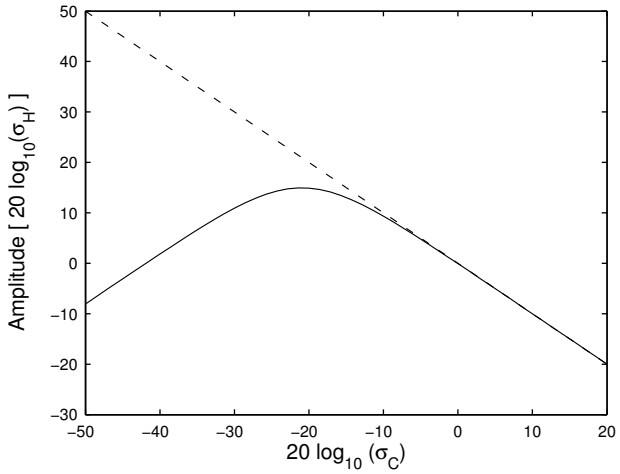
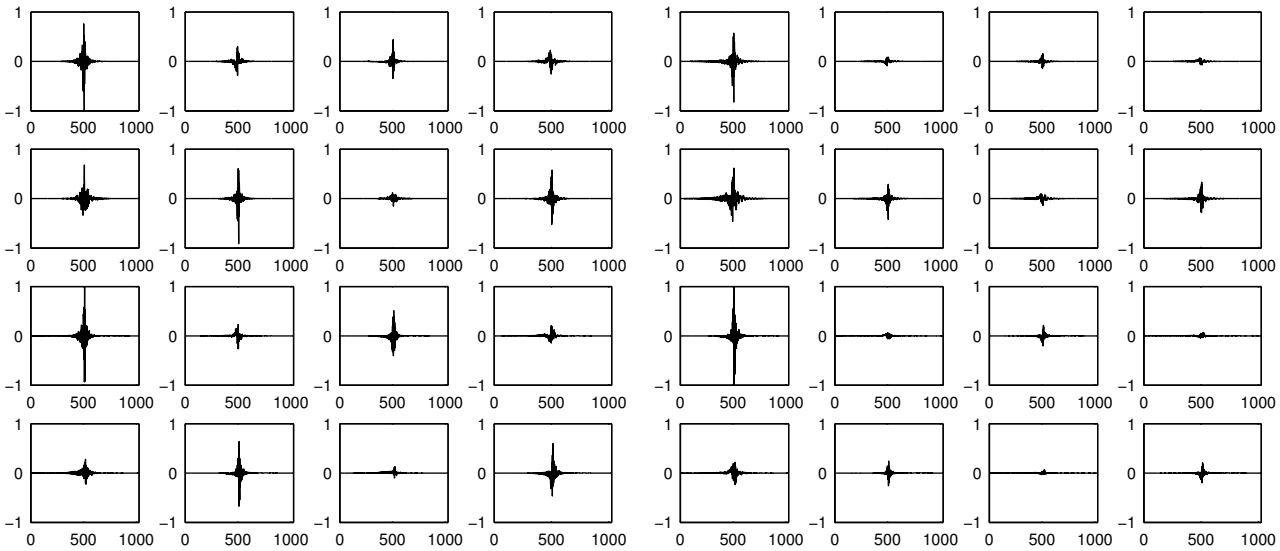


Figure 4.7: Influence of regularisation of singular values. - - - - $\sigma_{\mathrm{IF}} = \frac{1}{\sigma_{\mathrm{C}}}$, ——— $\sigma_{\mathrm{TIF}} = \frac{\sigma_{\mathrm{C}}^2}{\left(\sigma_{\mathrm{C}}^2+0.008\right)\sigma_{\mathrm{C}}}$

Figure 4.8 shows the resulting IRFs when the filters are normalised such that the largest peak is ± 1. By implementing diagonal preconditioning in the analog domain, the amplitude of the IRFs are fairly similar, resulting in better use of the dynamic range of the transducers, whereas when implemented in the digital domain, the magnitude of the IRFs suffer as they are required to compensate for the poor sensitivities induced into the system and shown in Equation 4.30.

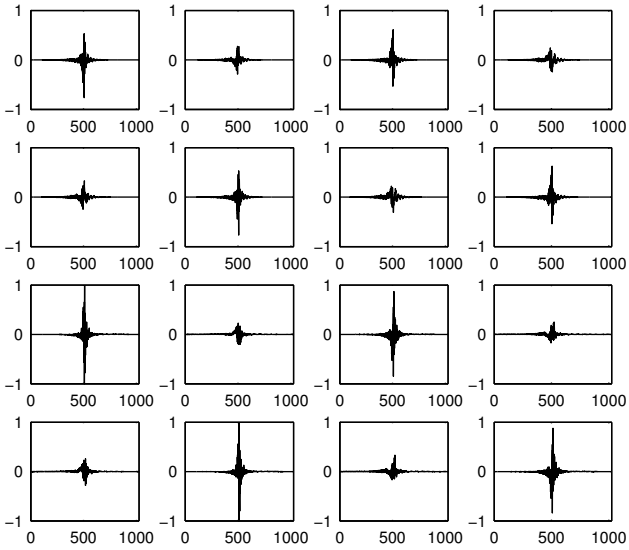
Figure 4.8c and Figure 4.8d shows the impulse response of the inverse filter designed to compensate for a system incorporating sensitivity compensation. Figure 4.8c shows the singular values for the inverse filter that was designed using the channel response incorporating sensitivity compensation, whilst Figure 4.8d shows the singular values for the inverse filter developed using the poorly scaled system and scaled to suit the system that has incorporated sensitivity compensation. The inverse filter developed using the poorly scaled system is observed (Figure 4.8) to make poor use of the channels compared to the inverse filter designed when the system has incorporated sensitivity compensation. Thus if the sensitivities of the transducers are adjusted, the inverse filter should be re-calculated rather than simply scaling the input and output to the inverse filter.

Figure 4.9 shows the IRFs of the entire system from the desired signal, $\mathbf{u}(z)$, to the received signal, $\mathbf{w}(z)$, using the filters shown in Figure 4.8. The ideal impulse response would be one that has a large central peak in the signal waveforms on the diagonal (referred to as *signal level*), with small side lobes (referred to as *signal quality*), whilst the waveforms on the off diagonal are completely flat (referred to as the *level of cross-talk*). In Figure 4.9a

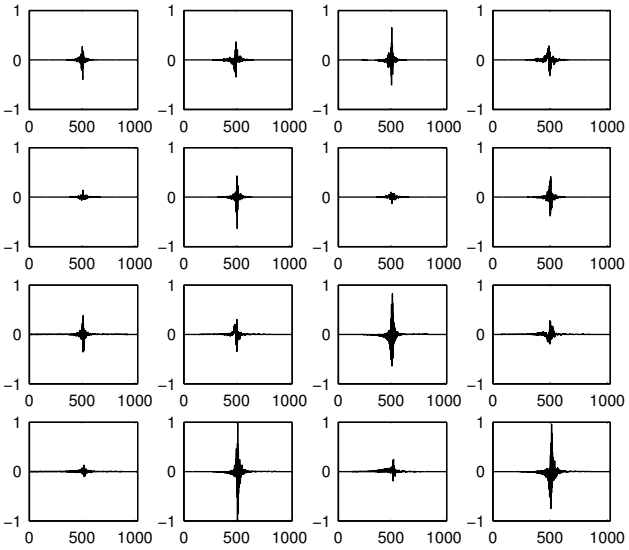


(a) No preconditioning, $\mathbb{H}\left\{\mathbf{C}_\mathrm{p}\right\}$

(b) Digital preconditioning, $\boldsymbol{\alpha}_\mathrm{g}\mathbb{H}\left\{\boldsymbol{\beta}_\mathrm{g}\mathbf{C}_\mathrm{p}\boldsymbol{\alpha}_\mathrm{g}\right\}\boldsymbol{\beta}_\mathrm{g}$



(c) Analog preconditioning, $\mathbb{H}\left\{\boldsymbol{\beta}_\mathrm{g}\mathbf{C}_\mathrm{p}\boldsymbol{\alpha}_\mathrm{g}\right\}$



(d) No preconditioning, scaled, $\boldsymbol{\alpha}_\mathrm{g}^{-1}\mathbb{H}\left\{\mathbf{C}_\mathrm{p}\right\}\boldsymbol{\beta}_\mathrm{g}^{-1}$

Figure 4.8: The impulse responses of the filters for $\kappa = 0.008$. The unit on the x-axis is samples. In these figures, the subplot at row i, column j corresponds to the IRF of the filter between virtual source j and transmitter i. These impulse responses have been normalised such that the largest peak value of each filter is ± 1.

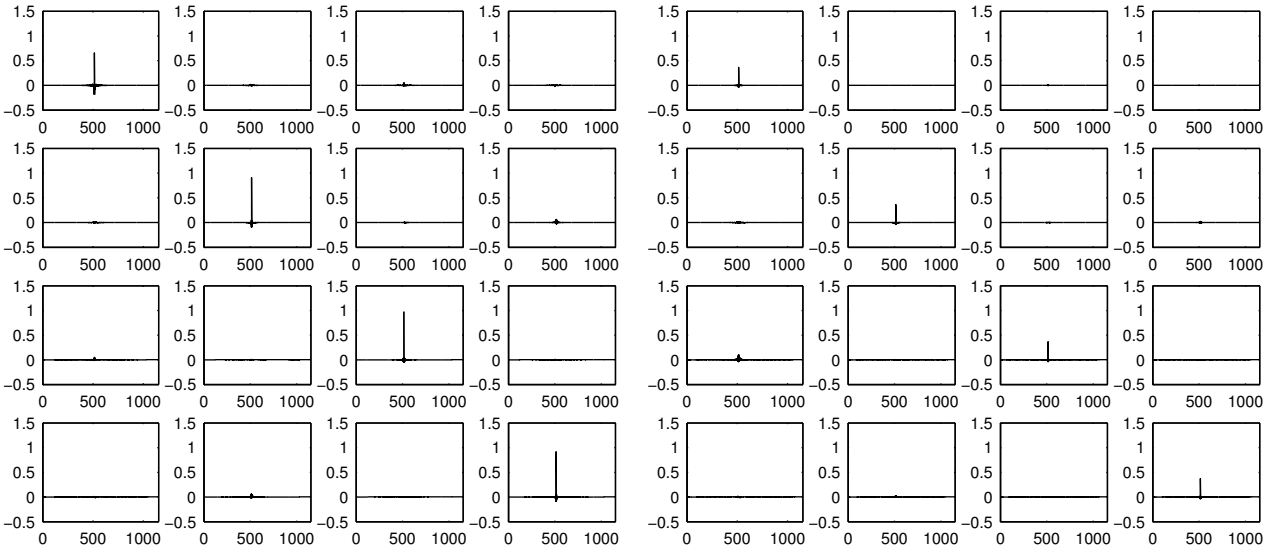
it is observed that because of the effort required to transmit to receiver 1, the regularisation has reduced the quality of the response at receiver 1, and also resulted in some cross-talk. When the sensitivities obtained using Algorithm 1 are used (Figures 4.9b and 4.9c), the magnitude and quality of the pulses are more consistent for each waveform on the diagonal, and there is also a reduction in the relative magnitude of the cross-talk. However, the implementation of diagonal preconditioning in the digital domain (Figure 4.9b), is shown to have obtained these improvements at the cost of reducing the signal levels.

Figure 4.9d shows the entire system response for the system using an inverse filter developed using the poorly scaled system and scaled to suit the system that has incorporated sensitivity compensation. It can be observed that the performance of this filter has greater cross-talk, uneven signal levels, and poorer signal quality than that shown in Figure 4.9c, being the filter designed for the properly scaled system.

Figure 4.10 shows the frequency response functions (FRFs) of the entire system from the desired signal to be received, $\mathbf{u}(z)$, to the actual signal received, $\mathbf{w}(z)$, using the filters shown in Figure 4.8a and 4.8c. The system response of Figures 4.8b and 4.8d have not been included, as they have very similar spectra to the filters shown in Figures 4.8c and 4.8a respectively. Comparing Figures 4.10a and 4.10b, the frequency response at the first receiver is noticeably improved with little change observed in the cross-talk cancellation, observable in the off-diagonal FRFs.

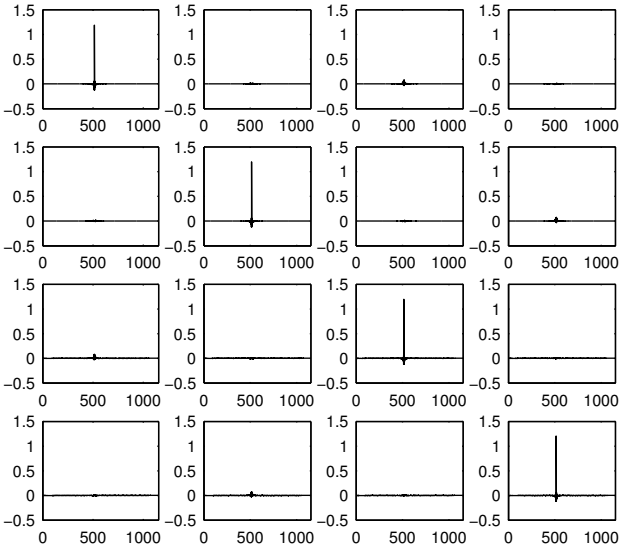
4.4 Conclusion

It has been demonstrated that the choice of sensitivities used within the amplifying stages of an acoustic system can have a significant influence on the performance of a system that incorporates a Tikhonov inverse filter. Unlike single channel systems, setting the sensitivities of the transducers to their maximum value for multi-channel systems does not always maximise the coherence between the input and output of the entire system consisting of the inverse filter, the sensitivities and the electro-acoustic system. An algorithm has been presented that generates a set of sensitivities that can compensate for poor combinations of receiver and transmitter sensitivities. In order to validate the algorithm, data from a simulation conducted by Kirkeby et al. [1998] was altered to emulate a system with a poor selection of transducer sensitivities. The algorithm was shown to obtain a desirable set of compensation gains that improved the performance of the system. The use of the compensation was investigated in both the analog and digital domains. The application of compensation sensitivities was shown to improve the quality of the received signal, however if the gains were applied in the digital domain,

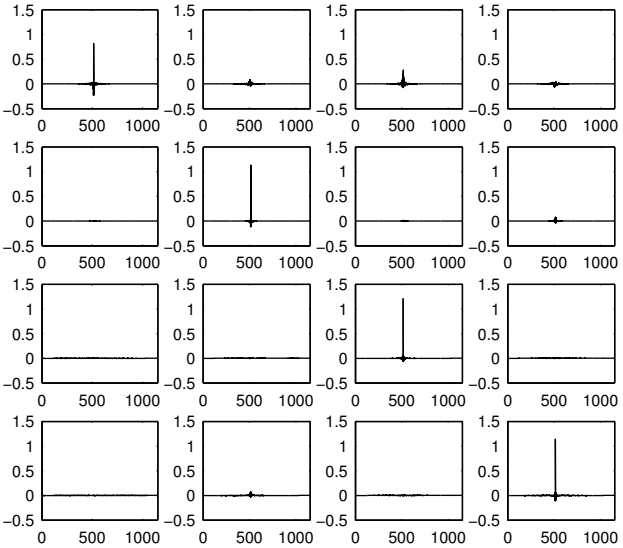


(a) No preconditioning, $\mathbb{H}\left\{\mathbf{C}_\mathrm{p}\right\}$

(b) Digital preconditioning, $\boldsymbol{\alpha}_\mathrm{g}\mathbb{H}\left\{\boldsymbol{\beta}_\mathrm{g}\mathbf{C}_\mathrm{p}\boldsymbol{\alpha}_\mathrm{g}\right\}\boldsymbol{\beta}_\mathrm{g}$

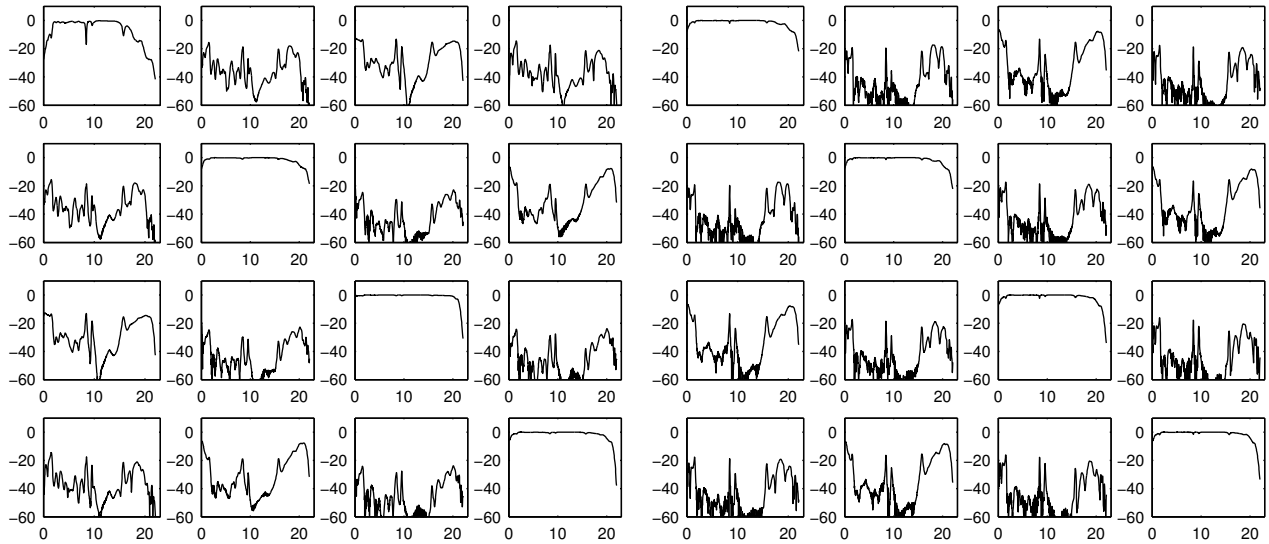


(c) Analog preconditioning, $\mathbb{H}\left\{\boldsymbol{\beta}_\mathrm{g}\mathbf{C}_\mathrm{p}\boldsymbol{\alpha}_\mathrm{g}\right\}$



(d) No preconditioning, scaled, $\boldsymbol{\alpha}_\mathrm{g}^{-1}\mathbb{H}\left\{\mathbf{C}_\mathrm{p}\right\}\boldsymbol{\beta}_\mathrm{g}^{-1}$

Figure 4.9: The impulse responses of the complete system for $\kappa = 0.008$. The unit on the x-axis is samples. In this figure, the subplot at row i, column j corresponds to the IRF of the entire system between virtual source j and receiver i.



(a) No preconditioning, $\mathbb{H}\{\mathbf{C}_\mathrm{p}\}$ (b) Analog preconditioning, $\mathbb{H}\{\boldsymbol{\beta}_\mathrm{g}\mathbf{C}_\mathrm{p}\boldsymbol{\alpha}_\mathrm{g}\}$

Figure 4.10: The frequency responses of the complete system for $\kappa = 0.008$. The unit on the abscissa is kHz, and the unit on the ordinate is dB. In these figures, the subplot at row i, column j corresponds to the FRF of the entire system between virtual source j and transmitter i.

the magnitude of the signal was reduced. If the compensation sensitivities were used to adjust the sensitivities of the transducers, it was shown that the inverse filter should be re-calculated rather than simply scaling the input and output to the inverse filter.

In the following chapter, the Tikhonov inverse filter design is used in conjunction with digital communication systems. A description of the experiment and simulation performed to implement the system is described in Chapter 5 and the theory and results from the experiment are presented in Chapter 6.

5 Experiment and Simulation

5.1 Overview

In order to investigate the application of the Tikhonov inverse filter to acoustic communication systems, an acoustic experiment was conducted in air. During the experiment it was found that because of the large number of variables that could be altered for each experiment, the results were insufficient to provide any conclusive outcome concerning the performance of the filter designs examined. The experiment was replicated in a model based simulation which could be executed more quickly than the air experiment and allow for a greater range of parameters to be tested in a shorter period of time. The purpose of this chapter is to provide a description of the experiment and simulation undertaken. The experimental apparatus, computing architecture, and computer code are presented and discussed. The details concerning the theory and outcomes are presented in Chapter 6.

5.2 Inverse filtering performed in a sound channel

5.2.1 Introduction

In order to compare the implementations of various inverse filter designs, an experiment was performed in an air waveguide. An acoustic model consisting of a static medium bound between two surfaces was used by Roux et al. [1997] and Roux and Fink [2000] in an experiment which examined the focusing of time-reversal. The results from the model provided insight into how time-reversal might perform in a shallow underwater acoustic environment since that environment also consists of a medium (water) bound between two surfaces: the sea floor and the air-water interface, and may be considered static for short periods of time.

In this research, a sound waveguide was used for the experiments, and the environment was bound by three surfaces. Since the waveguide is bounded by three surfaces, there exist more transmission paths than that for two

surfaces. The performance of some inverse filter designs is known to improve with the number of transmission paths (see Section 3.2.1), so it was decided to make the array span a plane rather than a line to make the environment more challenging for the inverse filters.

The medium used in the experiment was air, whilst that used in the model by Roux and Fink [2000] was water. In Section 3.2.1 it was discussed how the performance of time-reversal was primarily dependent on the modes created between the sea surface and the sea floor. Because the dimensions of the waveguide are vastly different from the depth of the sea, and the speed of sound in air is different from that in the sea, the frequency of transmission required to excite the modes is also very different. In order to relate the performance of the inverse filter from these experiments to that expected from a static shallow underwater environment, it was ensured that the frequency of operation in the waveguide was well above the cut-on frequency of the first mode, being defined as [Elliott, 2001]

$$f_c = \frac{c}{2L_y} \tag{5.1}$$

where c is the speed of sound, and L_y the maximum cross-sectional length of the waveguide. The waveguide width was around 400 mm and thus assuming a speed of sound of 343 m/s, the cut-on frequency was 430 Hz.

Whilst the environment has been shown to vary considerably from a shallow underwater waveguide, it has been considered as a reasonable environment in which to perform preliminary testing of the proposed algorithms before conducting experiments in the sea. An advantage of using the waveguide was that the experiment could be performed in a laboratory. The waveguide also proved to be an easy environment to configure; having a fast turn-around for obtaining results to be able to continually alter and develop the inverse filter designs which was the focus of this research.

5.2.2 Experiment configuration

A schematic of the equipment used for the experiment is shown in Figure 5.1. The configuration consists of microphone and speaker arrays located at each end of the waveguide, with each microphone placed directly in front of a speaker. The microphones were placed directly in front of the speakers so that future experiments using the environment might be able to use the principle of reciprocity (described in Section 3.2.1). In this experiment, only one speaker array and one microphone array were used, each at different ends of the waveguide. Because the dSPACE DS1104 controller card used for the experiment only had 8 A/D and 8 D/A converters, two switch boxes were used to select the speakers and microphones to play and record. The first switch box was connected to the D/A converters to select between the signals

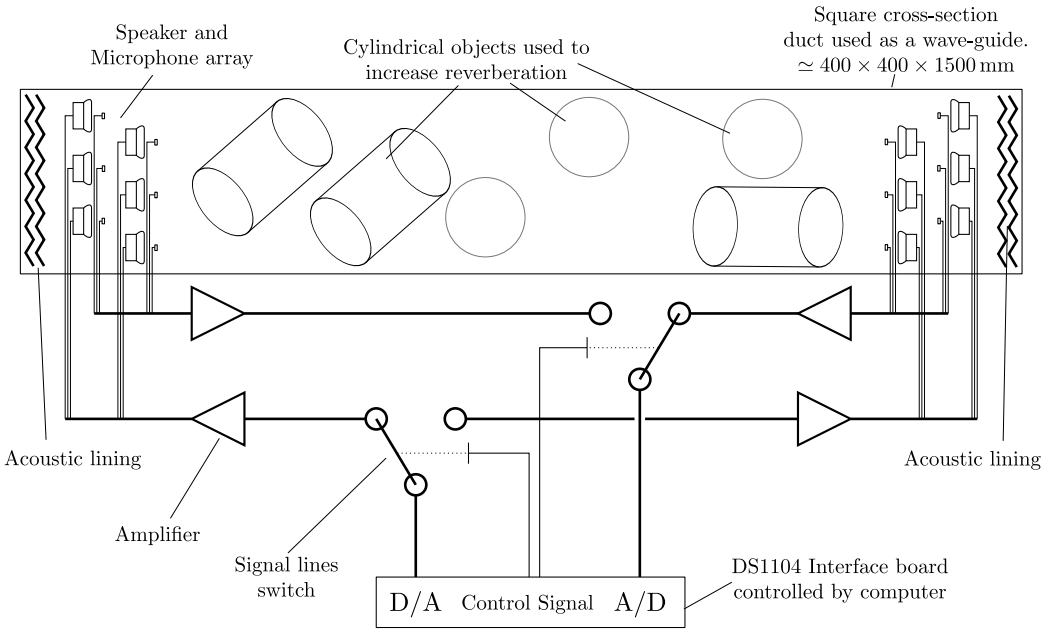


Figure 5.1: Experiment schematic

coming from the arrays at either end of the waveguide. The second switch box was connected to the A/D converters to select from which array to record the microphone signals. Amplifiers were incorporated between the speakers and microphones to adjust the signal level for each individual speaker or microphone.

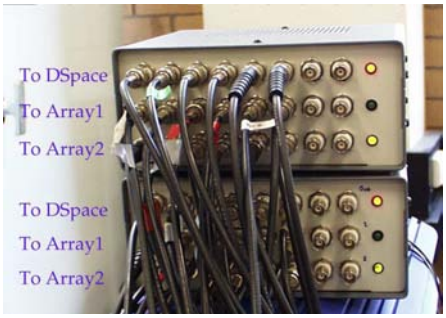
Photographs of the equipment used in the experiment are shown in Figure 5.2. The equipment shown are the waveguide, one of the arrays, and one of the switch boxes. The arrays consisted of six speaker and microphone pairs arranged on a 3 by 2 grid with a grid spacing of 55 mm. The array was mounted on a metal frame and separated from the end of the waveguide using sound absorbing material (shown in Figure 5.2a) to make the waveguide approximately acoustically unbounded lengthwise. Various cylindrical objects were placed inside the waveguide to increase the reverberation within the environment.

5.2.3 Characteristics of the system components

In order to keep costs to a minimum, the components used in the system configuration were chosen based on affordability rather than quality. Given the low cost of the speakers and microphones, the performance of the devices were expected to vary from that of higher quality devices. As the objective of the inverse filter designs is to compensate for an acoustic channel with an unknown system response, the characteristics of the microphones and

(a) The waveguide

(b) The microphone and speaker array



(c) Switch box - Front view

(d) Switch box - Back view

Figure 5.2: Images of the waveguide equipment

speakers would also be included as part of the system that is compensated. However, the use of poor quality components could result in the inverse filter requiring considerable effort to compensate for the characteristics of the device rather than the channel. To assist in understanding the effort required by the inverse filter to compensate for devices, an analysis of the various components in the experiment was conducted.

5.2.3.1 Speaker amplifier characterisation

The speaker amplifiers used in the experiment were of a type typically used in car audio systems. In this examination of the amplifier, the gain of the amplifiers was set so that a 1 kHz tone at 2 V_{rms} resulted in a 2 V_{rms} output signal. The measured frequency and phase response of the amplifier using this gain is shown in Figure 5.3 with no loading. The frequency response of the amplifiers fluctuate by less that 0.6 dB over the range 500 Hz - 50 kHz, and the phase follows a straight line corresponding to a delay of 2 μs. The frequency response obtained for the amplifier is exceptional such that it is questioned if the variation observed might also be the result of the measurement equipment. The result shows that the amplifier imposes very little distortion on the signal.

5.2.3.2 Speaker characterisation

The equipment used to examine the characteristics of the speaker is shown in Figure 5.4. A speaker was placed in an anechoic chamber with sound absorbent material surrounding the speaker. A high quality microphone (Brüel & Kjaer) was placed 283 mm away from the speaker, on axis. The frequency and phase response between the signal provided to the speaker and the signal received from the microphone were measured using a spectrum analyser whilst transmitting white noise through the speaker. A delay of 839.2 μs was applied to the measured speaker signal to maximise the coherence of the measurement by counteracting the delay from the travel of the sound waves through the air.

The magnitude, phase and coherence measurements for all the speakers are shown in Figure 5.5, where the magnitude of the response for each speaker has been normalised such that the average between 8 kHz and 13 kHz is 1. The amplitude response is relatively flat between 5 kHz and 15 kHz, after which the response rises, and after 22 kHz a number of nulls are evident. The coherence is very good except surrounding the nulls. Although the amplitude and phase response varies quite significantly, the response between the speakers is similar up until around 40 kHz.

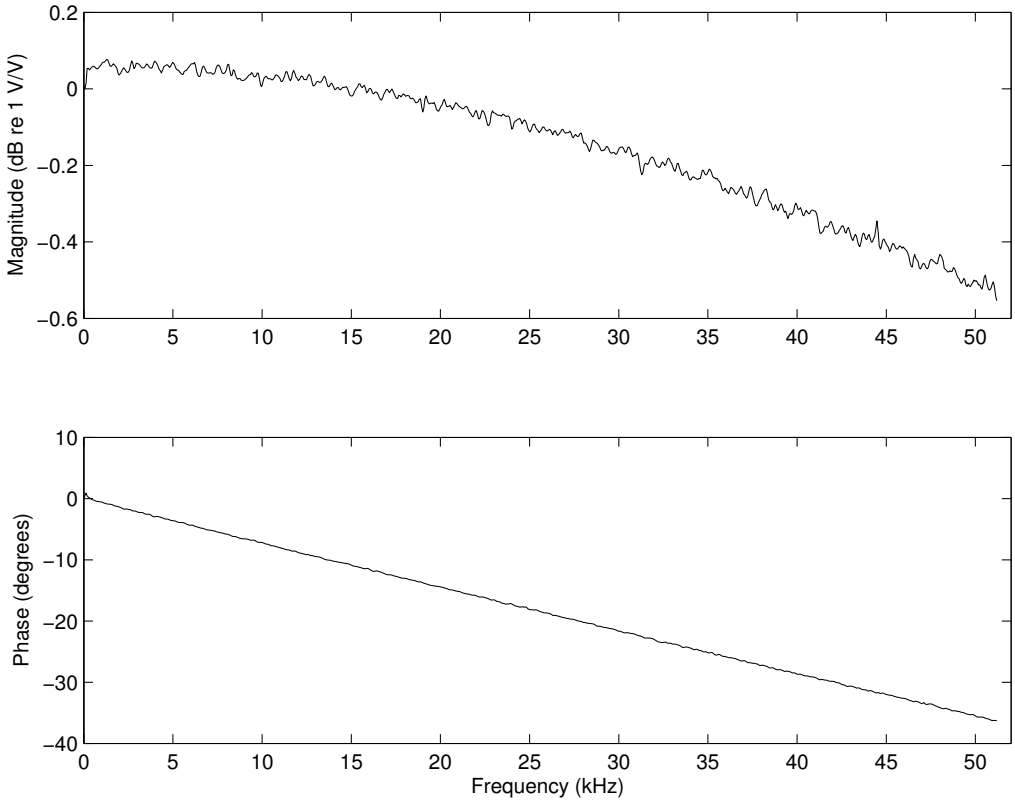


Figure 5.3: Measured bode plot for speaker amplifiers.

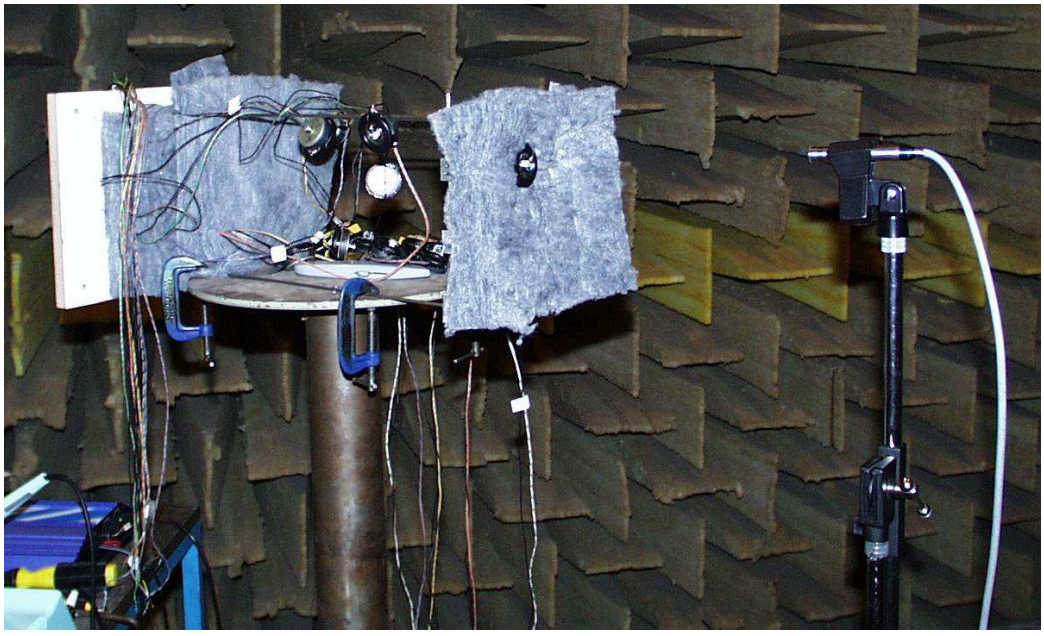


Figure 5.4: Equipment setup used to characterise the speakers.

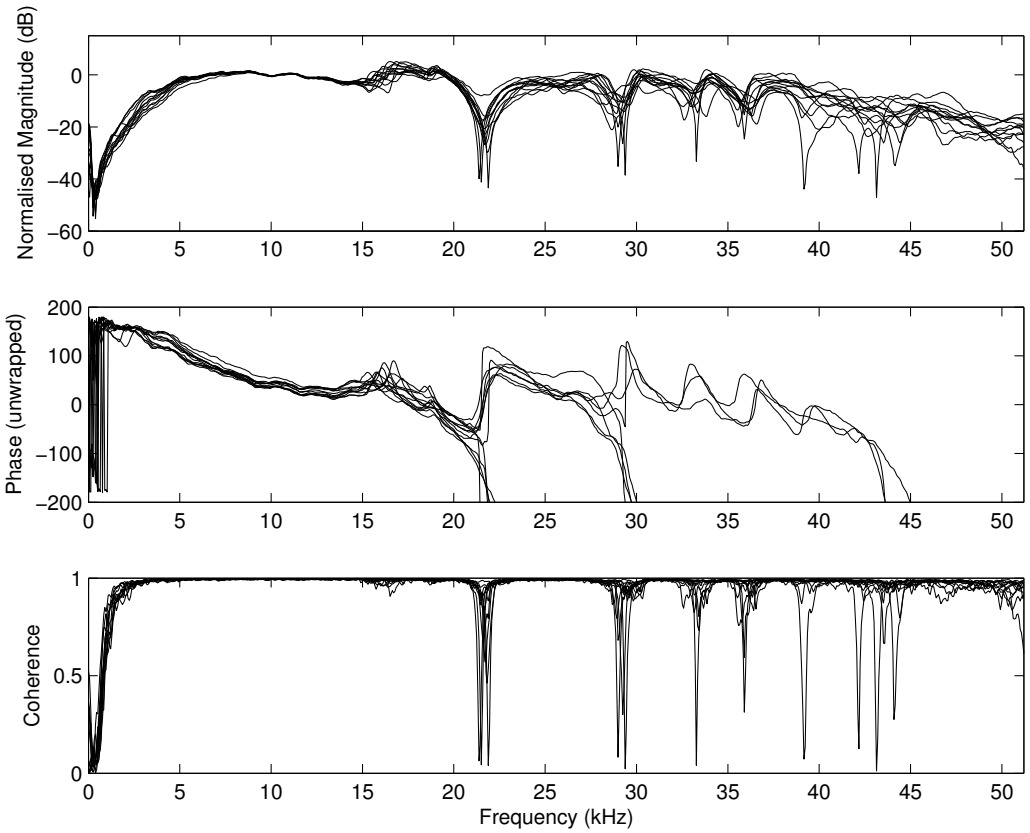


Figure 5.5: Magnitude, phase and coherence measurements for the speakers. The magnitude of the response for each speaker has been normalised such that the average between 8 kHz and 13 kHz is 1.

5.2.3.3 Microphone amplifier characterisation

The amplifiers used to convert the signal from the microphones used in the experiment were custom built and developed by the School's electronic workshop. The amplifiers provided three levels of gain, 10, 100, and 1000. The frequency and phase responses for these amplifiers are shown in Figure 5.6. Between 300 Hz and 20 kHz, the variation was limited to 1.7 dB, and the phase plots were observed to be almost linear with frequency between 300 Hz and 15 kHz, corresponding to group delays of 2.04 μs, 4.66 μs and 6.95 μs for the gains 10, 100 and 1000 respectively. Whilst the response of the amplifier is not ideal, the characteristics should be sufficiently well behaved for the inverse filters to overcome.

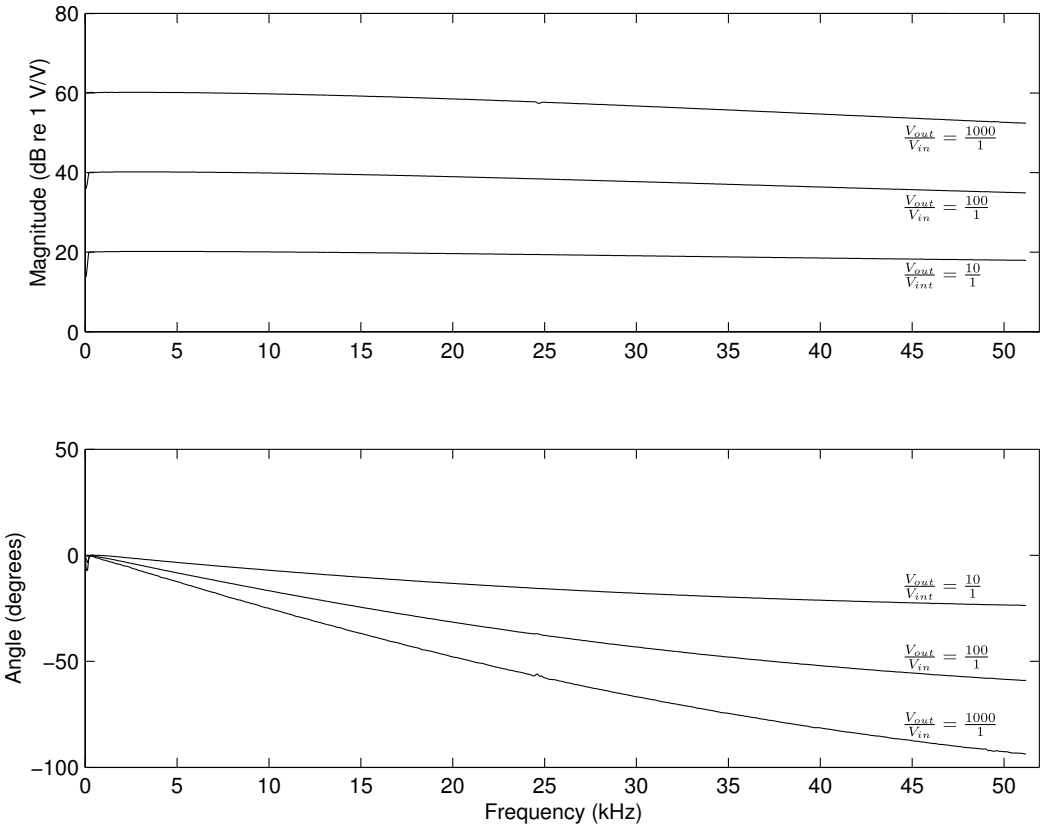
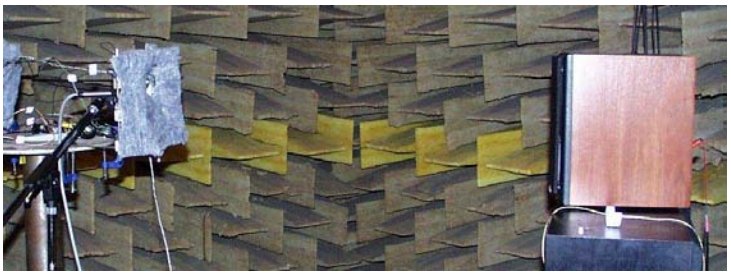


Figure 5.6: Measured bode plot showing the magnitude and phase for a microphone amplifier.

5.2.3.4 Microphone characterisation

In this section, the frequency response of the array microphones used in the experiment is examined. The equipment used to examine the array microphone characteristics is shown in Figure 5.7. A high quality 1/4" microphone (Brüel & Kjaer) was placed as close as possible to one of the array microphones. A loudspeaker was then placed on axis around 1 m from the microphones.

The frequency and phase response between the high quality microphone and the array microphone was measured whilst emitting white noise from the loudspeaker. The resulting bode plot obtained from the measurements is shown in Figure 5.8 for all the microphones, where the magnitude of the response for each microphone has been normalised such that the average between 5 kHz and 10 kHz is 1. The amplitude of the frequency response is observed to be similar over all the microphones. The amplitude response between 300 Hz and 15 kHz reduces smoothly and the phase also decreases in a roughly linear manner. At 16 kHz, the response of the microphones undergoes a critical change, and then settles again from 17 kHz through to



(a) View of loudspeaker, and microphone separation

(b) Close-up view of microphone co-location

Figure 5.7: Equipment setup used to characterise the microphones.

25 kHz. The fluctuation observed at 16 kHz could be attributed to the fact that the centres of the two microphones are separated by a distance of around 10 mm. When the spacing between the two microphones is greater than half the wavelength some variations between the responses should be expected as the measurements will be 180 degrees out of phase at this point. Assuming a speed of sound of 343 m/s, the 180 degree shift observed at 16 kHz, has a corresponding half wavelength of 10 mm, and is considered to be the reason for the fluctuation observed. Apart from this large fluctuation the frequency response appears to be consistent, and the coherence is generally flat. Although the amplitude response of the microphone varies by up to 20 dB between 0 and 25 kHz, the variations are smooth, with the exception of the variation at 16 kHz, therefore it should not be too difficult using the inverse filter to compensate for these variations.

5.2.3.5 Concluding remarks on system component characterisation

From examination of all the components, it can be observed that the major causes of distortion in the system are the speakers and the microphones. If the operational frequency is between 5 kHz and 25 kHz, then any fluctuation in the inverse filter performance observed at 22 kHz could be attributed to the compensation required for the speaker characteristics. A spectral amplitude plot as a result obtained from one of the experiments conducted using the speakers and microphones is shown in Figure 5.9. The influence of the response of the speakers at 22 kHz is observable. However the variation at around 16 kHz seen in Figure 5.8 does not appear observable, indicating that the fluctuations observed in the microphone characterisation were indeed the result of the spacing between the microphones.

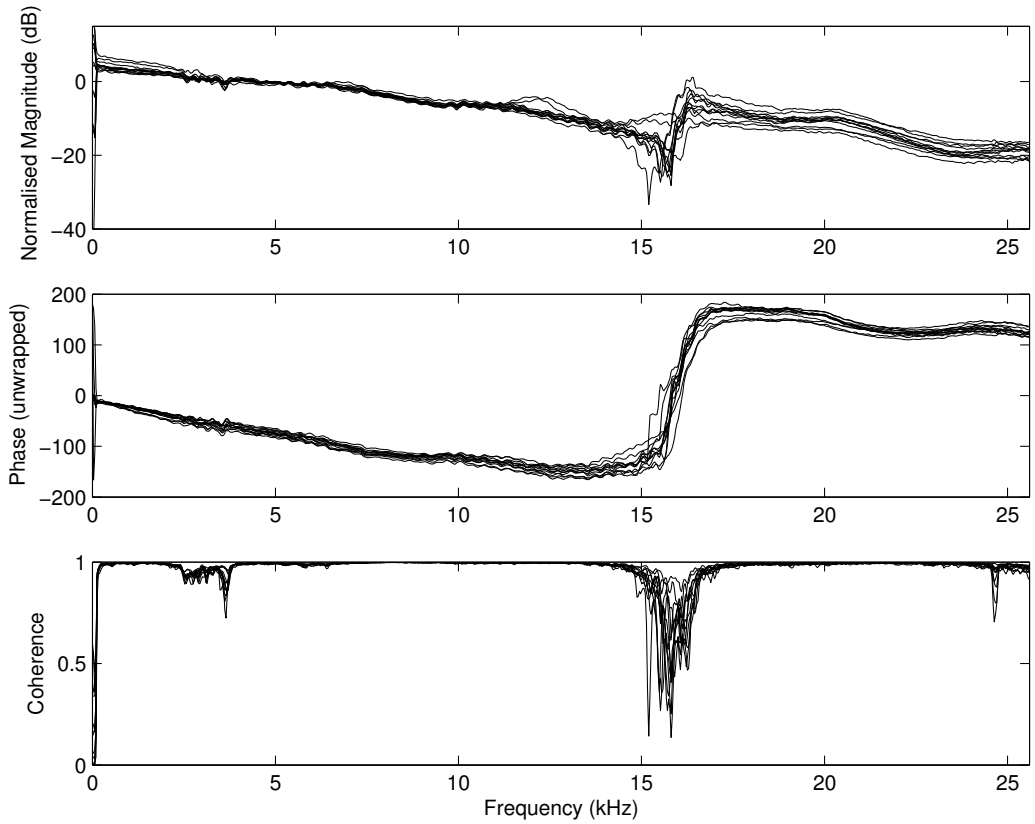


Figure 5.8: Magnitude, phase and coherence measurements for the microphones. The results have been normalised such that the average magnitude of each microphone measurement between 5 kHz and 10 kHz is 1.

5.2.4 The computer program code

To implement this experiment, a computer and a special programmable computer card were used to perform the signal processing parts of the experiment. The programmable computer card used was the dSPACE DS1104 R & D controller board. Whilst the dSPACE R & D board is designed to be used to perform real-time simulations of signal processing systems designed using MATLAB Simulink, it was found that the speed of the device was not sufficient for the sampling rates required when conducting the experiment. As a result, the dSPACE DS1104 device was used to only perform playback and recording of the analog signals, and the signal processing required was performed in MATLAB. To obtain the fastest sampling rate possible, the DS1104 device was programmed using a "hand-code" C-program in conjunction with a library provided by dSPACE that allows MATLAB to be able to easily communicate with the device. The program code developed to run on the DS1104 controller board, along with the MATLAB program code used

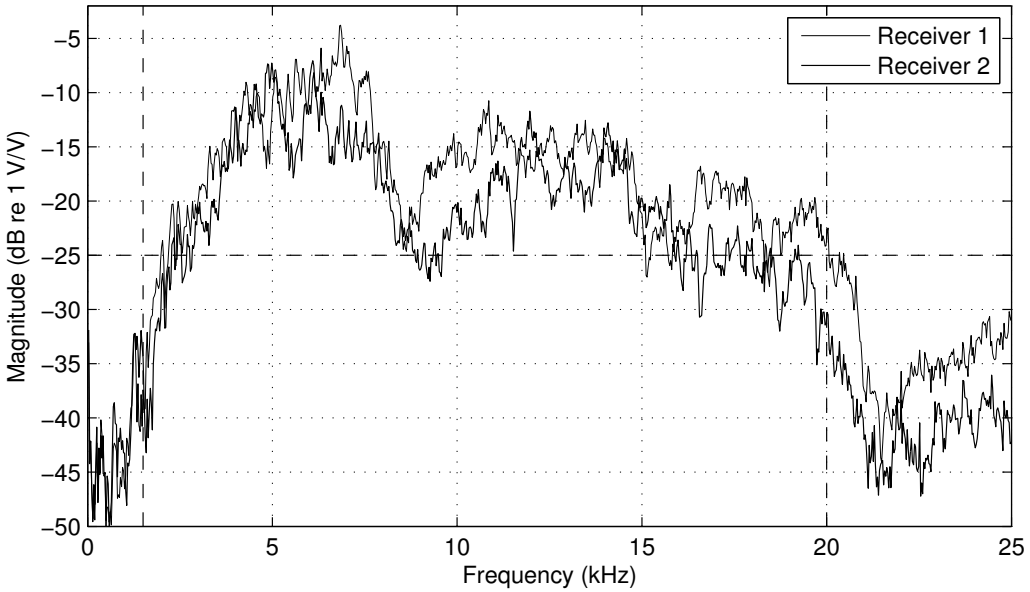


Figure 5.9: Average frequency response between six transmitters and two receivers from one of the experiments conducted.

to interact with the controller board, and perform the desired calculations is presented in Appendix A. A textural description accompanies the program listings to provide the reader with an understanding on how the programs perform the desired functions.

5.2.5 Experimental procedure

In order to conduct the experiment, the computer and dSPACE DS1104 were connected to the equipment as shown in Figure 5.1. The source code for the main scripts of the software used to run the experiment is presented in Appendix A.2. In addition to the main scripts, a number of *helper scripts* were developed for use by the main scripts to perform specific functions. Some of the helper scripts are specific to this experiment, whilst others have been re-used in the simulation discussed in Section 5.3. The scripts specific to this experiment are presented in Appendix A.2.4, whilst the functions that have been re-used have been placed in a *"thesis MATLAB library"* given in Appendix A.4.

To run the experiment, the code, `DuctExperimentDSPACEProgram.c`, was compiled using the compiler provided by dSPACE and then downloaded onto the controller card. The MATLAB script, `DuctExperimentCreate-TransmissionSignal.m`, was then executed to interact with the dSPACE DS1104 device to measure the impulse response of the transmission signals,

and generate the modulated signals required to transmit a desired binary sequence using the various inverse filter design schemes. The script, `DuctExperimentPlayAndPostprocess.m`, is then used to play the signal, and perform post-processing on the signal received to examine the performance of the inverse filter designs.

Transmission Signal Generation

The specific steps performed by the script, `DuctExperimentCreateTransmissionSignal.m`, to create the transmission signal are as follows:

1. Measure the channel impulse response functions

   The impulse response function is measured through the use of the function `GetIRFsDS1104.m` which creates a signal used to characterise the system and send the data to be played to the dSPACE controller. The function `GetIRFsDS1104.m` then makes the DS1104 controller card execute the routine `Play_Stepped_SIMO` on the dSPACE controller that plays the signal independently on each of the desired output channels, and records the response on all the inputs. The function `GetIRFsDS1104.m` then retrieves the signal and uses the MATLAB function, `spectrum`, to determine the set of impulse responses between each transmitter and receiver.

2. Convert the channel impulse response to base-band

   Whilst the inverse filters could be designed at pass-band, it was found that the inverse filter could also be designed with reduced complexity at base-band. This conversion was performed according to Equation 2.26.

3. Down-sample the channel impulse response for the design of the inverse filters

   Referring to Figure 2.10, if the inverse filters are designed at passband, the sampling frequency must be greater than $2(f_c + \frac{B}{2})$, where f_c is the carrier frequency and B the bandwidth. However, if the channel response is converted to base-band, then the sampling frequency of the base-band impulse responses can reduced to $2B$. By designing the inverse filter at a lower frequency rate, the number of computations can be reduced compared with designing the inverse filter at passband.

4. Design of the inverse filters

   The function `CreateInverseFilter.m` was used to design the inverse filters. The resulting inverse filter consists of a set of impulse responses between a set of virtual sources, and the transmitting devices, where the virtual sources are the signals desired to be received at the receiver

devices after a given time delay (required to make the inverse filter realisable). The function `CreateInverseFilter.m` can design the following filters:

No Filtering This filter type is effectively an identity matrix, and effectively transmits the signal desired to be on the i-th receiver on the i-th transmitter.

Time Reversal The time-reversal filter design employed used the reverse of the channel response between the i-th transmitter channel and the j-th receiver channel as the filter between the j-th virtual source, and the i-th transmitter channel. In this instance, the time-reversal filters have been normalised for each filter independently.

Tikhonov Inverse Filter Three variations in the design of the Tikhonov inverse filter were developed as part of this thesis. The designs are: *inverse by path* (SISO), *inverse by channel* (MISO) and *inverse by full response* (MIMO). These forms of implementation are described in Section 6.2.1, given by Equations 6.7, 6.8, and 6.9.

Stojanovic Two-Sided Filter During the time that the experiment and subsequent simulations where being conducted, Stojanovic [2005] published a paper that was considered to have contributed valuable inverse filter designs against which to compare the Tikhonov inverse filter. One of the designs implemented in this research was the two-sided filter. This filter design is described in greater detail in Stojanovic [2005], and has also been modified in this thesis to include regularisation (to improve performance) as discussed in Section 6.2.3. The equation for this filter is given by Equation 6.13.

5. Up-sample the inverse filter impulse responses

   Following the design of the inverse filters using the lower sampling rate, the inverse filters are up-sampled to that of the sampling rate used in the experiment.

6. Create the transmission signal

   A digital sequence is developed that consists of 1 followed by 499 random samples. This binary sequence is then mapped to a series of complex symbols according to the Phase-Shift-Keying (PSK) modulation using the function `BitSequenceToComplexSequence.m`. This sequence is then pre-pended with a lead-in sequence that is used to synchronise

the receiver to the time and phase of the signal. A raised cosine square-root spectral shaping filter is then designed using the function `RaisedCosineFrequencySpectrum.m`, which is used to convert the complex sequence of symbols to a base-band signal using the function `ComplexSequenceToSignal.m`. The theory concerning the generation of the communication signal as performed in this step is described in more detail in Section 2.2.4.

7. Filter the transmission through the inverse filter

   The base-band transmission signal is then used as the virtual signal for the first channel, and passed through each of the inverse filter designs to form a set of base-band signals to play through the transmitters.

8. Convert the signal to pass-band

   The base-band signal for the transmitter is converted to pass-band using the function `BasebandToPassband`, and normalised so the maximum signal level is unity. This conversion was performed according to the formula given in Equation 2.25.

Transmission and reception of the signal

The steps performed by the script, `DuctExperimentPlayAndPostprocess.m`, used to play and process the received signals are:

1. Play the transmission through the system.

   The pass-band transmission signal developed for each of the inverse filters was played whist recording the response using the function `RunDS1104MIMO.m`. This function was designed to transfer the play data from the MATLAB environment to the controller board and then trigger the board to run the routine `Play_MIMO`. After the routine `Play_MIMO` is finished, the function `RunDS1104MIMO.m` retrieves the recorded signals.

2. Convert the signal to base-band and estimate the time delay and phase of the signal

   The function `SignalPhaseEstimatorPassbandToBaseband.m` was developed to convert the signal from pass-band to base-band using the hetero-dyne technique described in Section 2.2.3. The distance between the transmitter and received signal results in the base-band received signals being delayed and shifted in phase from that of the transmitted base-band signal. To estimate the delay and the phase, a once-off delay and phase estimate was performed. The method used to estimate the delay was to find the peak magnitude of the signal in the range of one

symbol length after the time at which the signal rose above 40% of the received signal value. Given that the lead-in sequence consisted of a one followed by zeros, the phase was calculated from the phase at the location of the peak magnitude signal, and was used to correct the phase of the entire base-band signal.

3. Sample the signal at the correct sampling instances.

4. Run the adaptive filter algorithms on the received signals using the function `DuctExperimentRunAdaptiveTests.m`. This function performed the following:

   a) Applied an adaptive filter with no tap adjustment to observe the performance of a detector without the use of an adaptive filter.

   b) Apply the adaptive algorithms using a number of different step sizes at the symbol rate.

      The adaptive filters examined were the Zero Forcing algorithm (ZF), Least-Mean-Square algorithm (LMS), and the Recursive Least Square (RLS) algorithm, each of which is described in Section 2.2. It was desired that a very long bit sequence be used to examine the results for each filter after the filter-taps had appropriately settled, however memory limitations, and the high computational complexity resulted in a limited number of samples that could be transmitted. To be able to examine the likely performance of each filter, the adaptive algorithms were applied multiple times to a shorter bit sequence. The adaptive filters were first applied with a very large step size and then using the final state of the filter taps as the initial state of the next iteration of applying the adaptive filters using a smaller step size. The step size of the adaptive filter was incrementally reduced for all the algorithms except the RLS algorithm.

5.2.6 Results from experiment

Preliminary examinations of the ability to focus sound were presented at the 145th meeting of the Acoustical Society of America conference [Dumuid and Cazzolato, 2003].

Throughout the experiment, there were a number of parameters that could be adjusted, such as the inverse filter regularisation value, the selection of which sources and receivers to use, the carrier frequency, and the symbol rate. When adjusting these parameters, the result obtained also varied. A sample of the results obtained for each experiment run are shown in Figures 5.10 to 5.13.

Figure 5.10 shows the sampled signal at the target receiver and the signal sampled simultaneously at the non-target receiver. The results for an ideal inverse filter would show a scatter plot consisting of dots at the typical 4-PSK phase positions at the target receiver and a set of dots on the origin for the non-target receiver. For this particular set of results, it can be observed that all the inverse filters have a 4-PSK scatter plot at the target receiver. The Tikhonov inverse filter designed by full response MIMO has the smallest variation in the signal received at the non-target receiver, whilst the time-reversal result has the largest variation although it does have the largest signal amplitude at the receiver. If a PSK was desired to be transmitted to the non-target receiver, then it would be expected that the scatter shown would be added to the scatter of the received PSK, and a similar level of interference would be received on the target receiver. By considering the effect of adding the fluctuations of the non-target receiver to the target receiver it can be observed that none of the inverse filters would have exceptional performance for this set of results.

To compensate for inter-symbol interference that might remain in the received signals, a number of adaptive filters were applied to the sampled signals. The adaptive filter algorithms employed were the Zero Forcing (ZF) algorithm, Least-Mean-Square algorithm (MSE), and the Recursive Least Square (RLS) algorithm. The MSE and RLS algorithms were also examined using fractional spacing. The adaptive filters were applied several times to attain the best possible set of taps as described in Section 5.2.5. The results obtained using the various adaptive filter designs are presented in Figures 5.11 and 5.12. Figure 5.11 shows the final scatter plots of the received signal for each inverse filter design and corresponding adaptive filter algorithms, whilst Figure 5.12 shows the average error between the detected symbol and the filtered signal, after each recursive application of the adaptive filters. It can be observed that the use of an adaptive filter provides reduced symbol fluctuation regardless of the adaptive filter or inverse filter design used. Whilst the RLS algorithm is shown to provide the least symbol error for all the inverse filter designs, the average symbol error signal is of the same order of magnitude as that of the other designs.

In order to ensure that the adaptive filters operated properly, the filter taps for each sample processed in the adaption were stored. Figure 5.13 shows the magnitude of these filter taps for each sample. From these graphs, it is evident that the RLS algorithm performs the adaption more rapidly than the other adaptive filter designs.

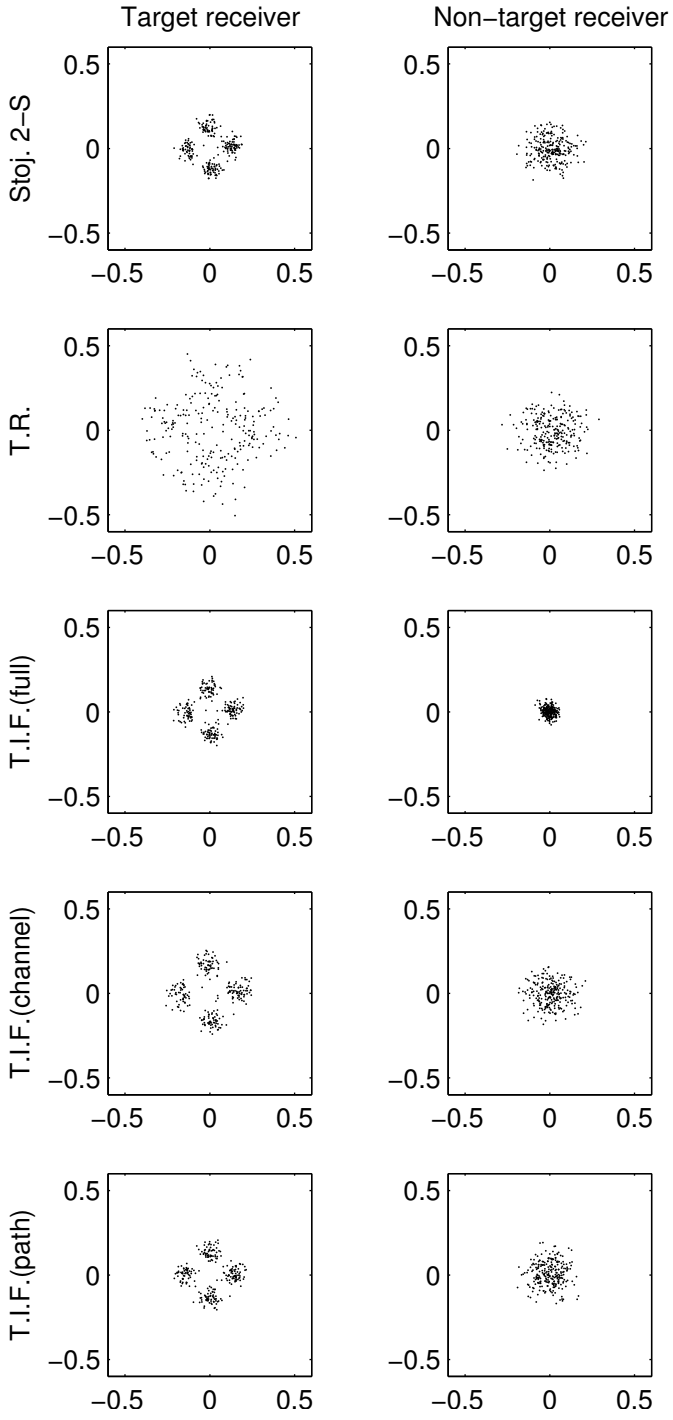


Figure 5.10: Scatter plot of the sampled signal at the target receiver and non-target receiver. The filters examined are the Stojanovic two-sided filter design [Stoj. 2-S], time-reversal [T.R.], Tikhonov inverse filtering using full [T.I.F. (full)], channel [T.I.F. (channel)], and path [T.I.F. (path)] structures.

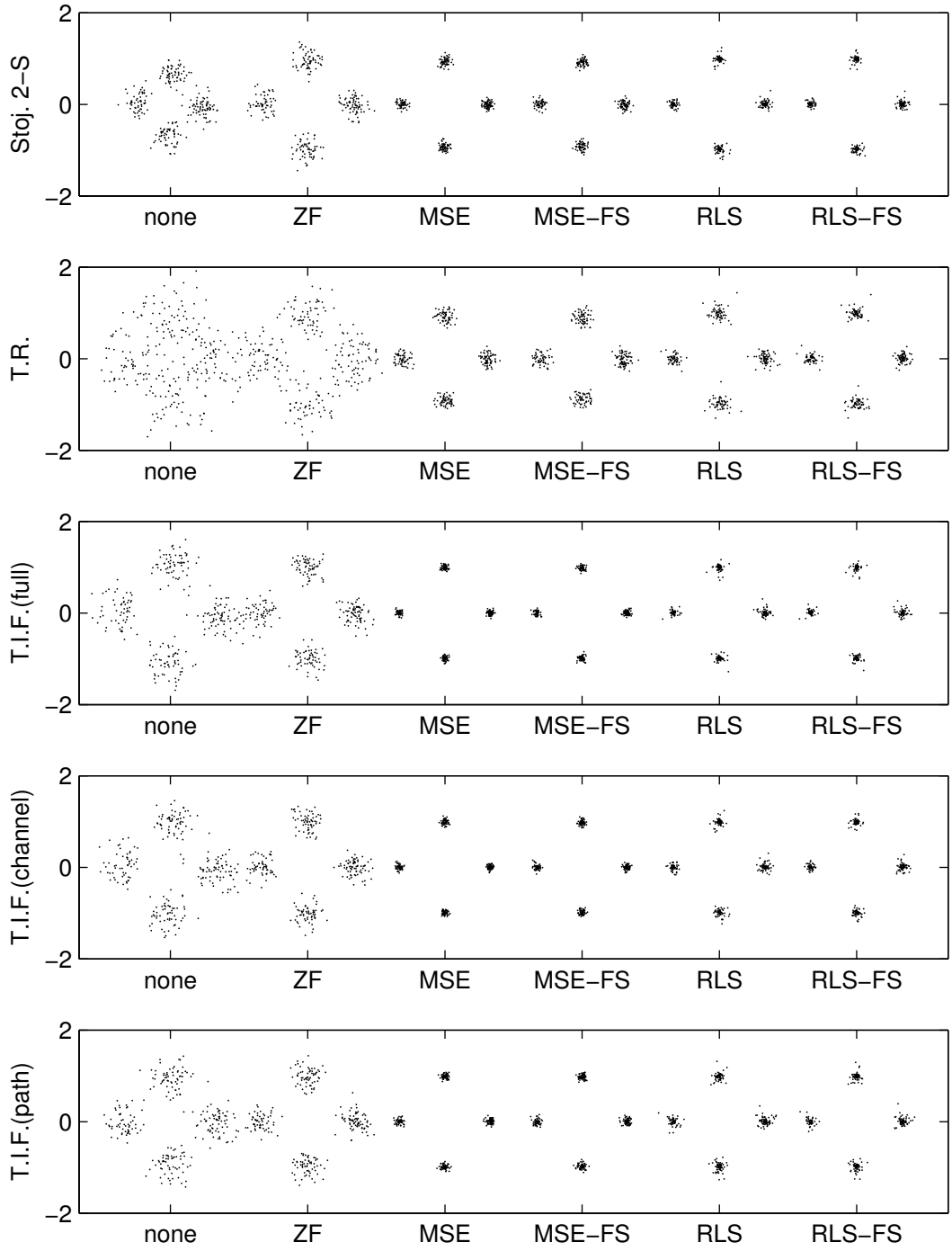


Figure 5.11: Scatter plots of the filtered target receiver signal after applying the different adaptive filtering algorithms to each of the signals received by the inverse filter designs. The adaptive filters are no filtering [none], the zero-forcing equaliser [ZF], the mean-square error equaliser [MSE], the fractionally-spaced mean square error equaliser [MSE-FS], the recursive least square error equaliser [RLS], and the fractionally-spaced recursive least square error equaliser [RLS-FS].

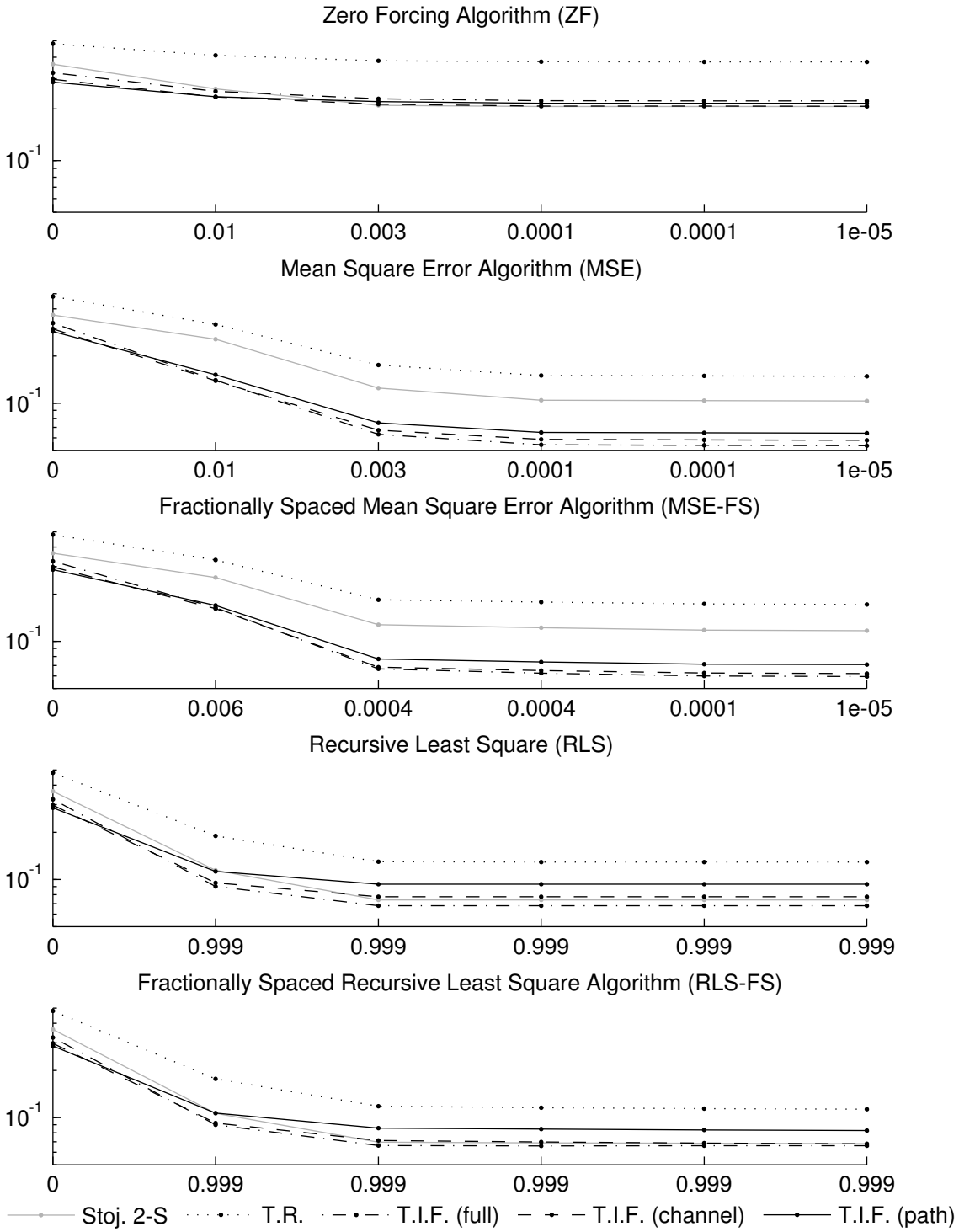


Figure 5.12: The history of the symbol error after each iteration of the various adaptive filter algorithms. The values shown on the abscissas are the step-sizes used for each iteration of the equalisers, and the ordinate value is the symbol error after the iteration. The step size was kept constant for each iteration of the RLS equalisers.

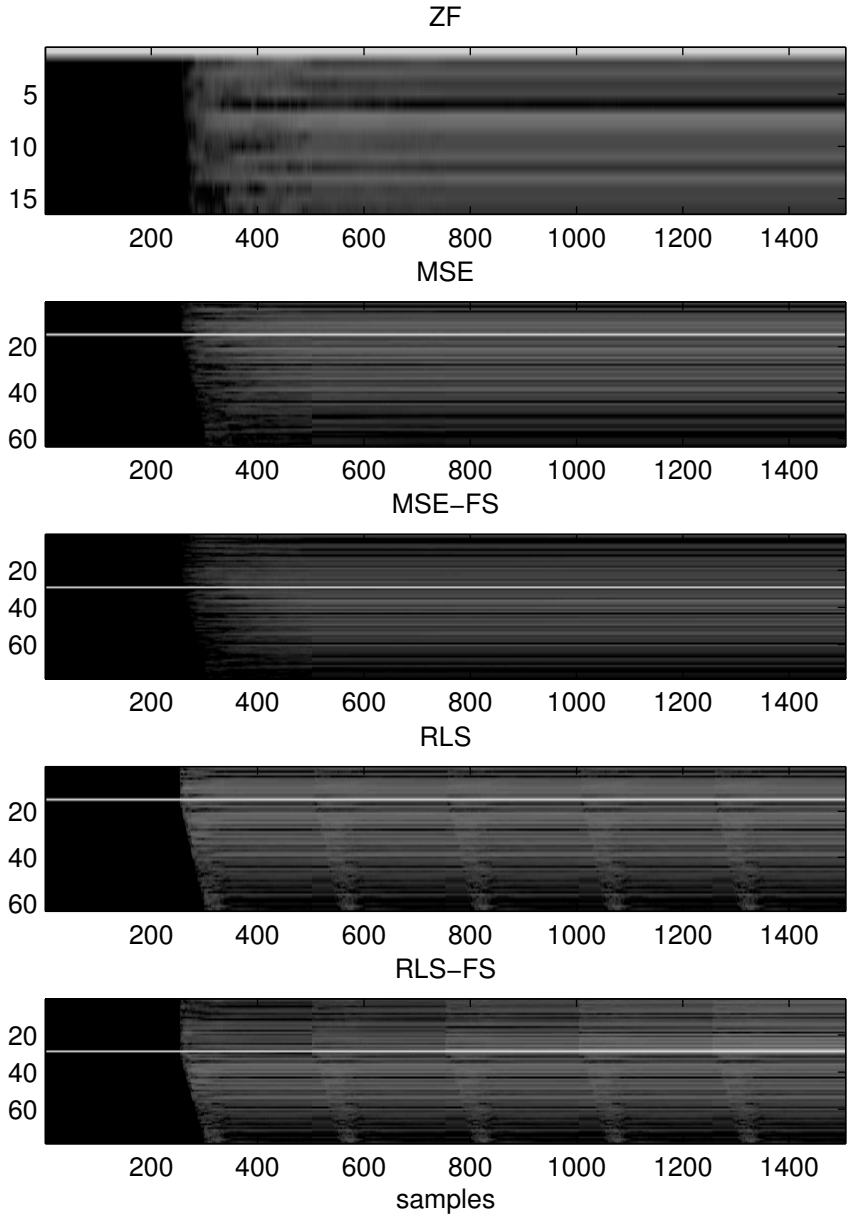


Figure 5.13: The magnitude of the filter tap after each sample of the adaptive filters for the Tikhonov inverse filter using the full structure.

5.2.7 Conclusion from the experiment

In this experiment, the ability to perform communication in a reverberant channel through the use of inverse filters was investigated. Whilst the results obtained demonstrated it was possible to achieve transmission over the channel, it was questioned if multi-channel communication could be performed. It was also shown that the performance of all the inverse filters examined could

be improved using adaptive equalisers. It was found that the variation and possible range of various parameters in the experiment made it difficult to obtain a clear understanding of the performance of each of the filter designs. In order to obtain more informative results concerning the performance of the inverse filter designs over the possible range of parameter values, a computer simulation was conducted that performed similar steps as the experiment, except that an estimate of the channel responses to the transmission signals were used. The computer simulation is described in greater detail in the following section.

5.3 Computer simulations

5.3.1 Introduction

In Section 5.2, an experiment was presented that implemented inverse filters in a digital communication system through an air waveguide using acoustic signals. The experiment contained many parameters that could be varied to alter the performance of the system. Due to the number of parameters that could be varied, it was difficult to identify the influence each parameter had on the results independently of the other parameters. In addition to the difficulty of characterising each parameter, since the experimental apparatus was not enclosed there was a high environmental noise and temporal variation of the channel response.

After measuring the channel impulse and generating the communication signal, it was possible to either emit the signal in the channel and measure the response or simulate the response. The simulated and experimental results were observed to be similar provided there were minimal disturbances from the environment. To examine the influence of each parameter it was decided to record a set of impulse responses from the channel, and develop a simulation that would measure the performance of the communication system using each inverse filter over a range of parameters. After implementing the simulation it was found that to test all the filter designs for a single carrier frequency, a single transmitter and receiver array arrangement and a single symbol rate over a range of 42 possible values of the regularisation parameter took an average time of 2.98 hours to execute. In order to examine the results over a greater range of parameters and in a timely manner, a distributed computing system was used.

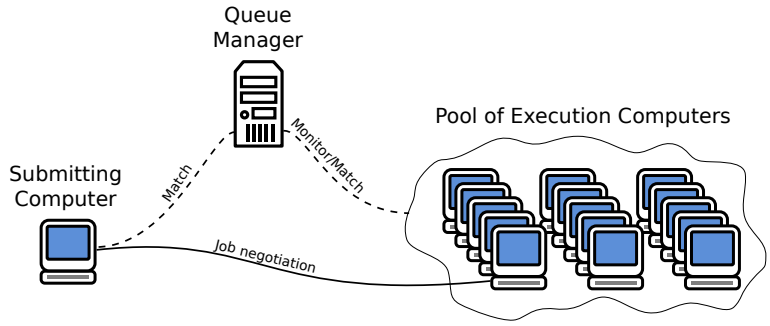


Figure 5.14: Schematic of the Condor distributed computing system. The pool of execution computers contained around 200 computers.

5.3.2 Implementation

5.3.2.1 The Condor system

Because of the number of computations required, it was decided that the simulations be performed using a pool of computers and managed by the distributed computing software, Condor. Figure 5.14 provides a schematic of the Condor system. The Condor system consists of a submitting computer, a queue manager, and a pool of execution computers. The pool of execution computers consisted of the undergraduate teaching suite computers from the Faculty of Engineering, Computer and Mathematical Sciences. Any machines not being used by the students at any point in time were available to the pool. The submitting computer contains a list of jobs desired to be executed. The queue manager monitors the computers in the pool and the jobs listed by each submitter and determines which computer in the pool matches the requirements for each job on the submitting computer. The queue manager then notifies both the submitting and executing computer of the match, and the submitting computer directly communicates with the computer to transfer the files required for the job, along with the commands to execute the job. After the executing computer has finished processing the job, the output files are returned to the submitting computer. The code used for the simulation is included in Appendix A.3.

5.3.2.2 MATLAB scripts that interact with Condor

To interact with the Condor system and control the simulation, several MATLAB scripts were developed. The schematic showing how these scripts generate the jobs, and process the results in presented in Figure 5.15. The main script that controls the simulation is `DuctSimulationManager.m`. This script calls the script `DuctSimulationSubmitJobsAndFetchResults.m` to create a set of files in a directory for each job and submit the job to the Condor

submitter machine. Subsequent calls to `DuctSimulationSubmitJobsAndFetchResults.m`, can be performed to re-submit the jobs (if the job failed or was cancelled because the computer was used by a student for other tasks) and fetch the results if available. The results fetched by `DuctSimulationSubmitJobsAndFetchResults.m` are then combined into a single variable to be saved and used when processing the results. The script used to process and display the results is `DuctSimulationResultViewer.m`.

The parameters that the simulation was used to investigate are:

- The carrier frequency.
- The inverse filter regularisation parameter, κ (for the designs that had a regularisation parameter).
- The symbol length.
- The transmission elements used (all 6, or various subsets).

The range of values over which these parameters were varied were defined in `DuctSimulationManager.m` in the variable `staticConfig.ranges`. Each job script tested the inverse filter designs over all the ranges of regularisation parameter, κ, for a specific carrier frequency, symbol length, and selection of transmitter elements. The script `DuctSimulationSubmitAndResultFetcher.m` submitted a job for each carrier frequency from the range of carrier frequencies tested. The symbol length and the transmitter elements were configured by the user by setting the variables, `simulationConfig.symbolLengthIndex` and `simulationConfig.arrangementIndex` prior to executing the script `DuctSimulationManager.m`.

5.3.3 The executing computer script

5.3.3.1 Overview

The source code to the scripts that were run on the executing computer is presented in Appendix A.3.3. The executing computer runs the batch file `CondorJobExecutor.bat` that connects to a remote network drive and executes a compiled version of the MATLAB script, `MainCondorJobScript.m`. The MATLAB script `MainCondorJobScript.m` loads the simulation parameters from the file `runparams.mat` and the channel impulse response and data sequence from the files `testIRFs.mat` and `bitseq2.mat`, stored on the remote network drive. The schematic of the simulation conducted is presented in Figure 5.16. The steps involved in the simulation consist of:

1. Generating a communication signal.
2. Generating the inverse filters.

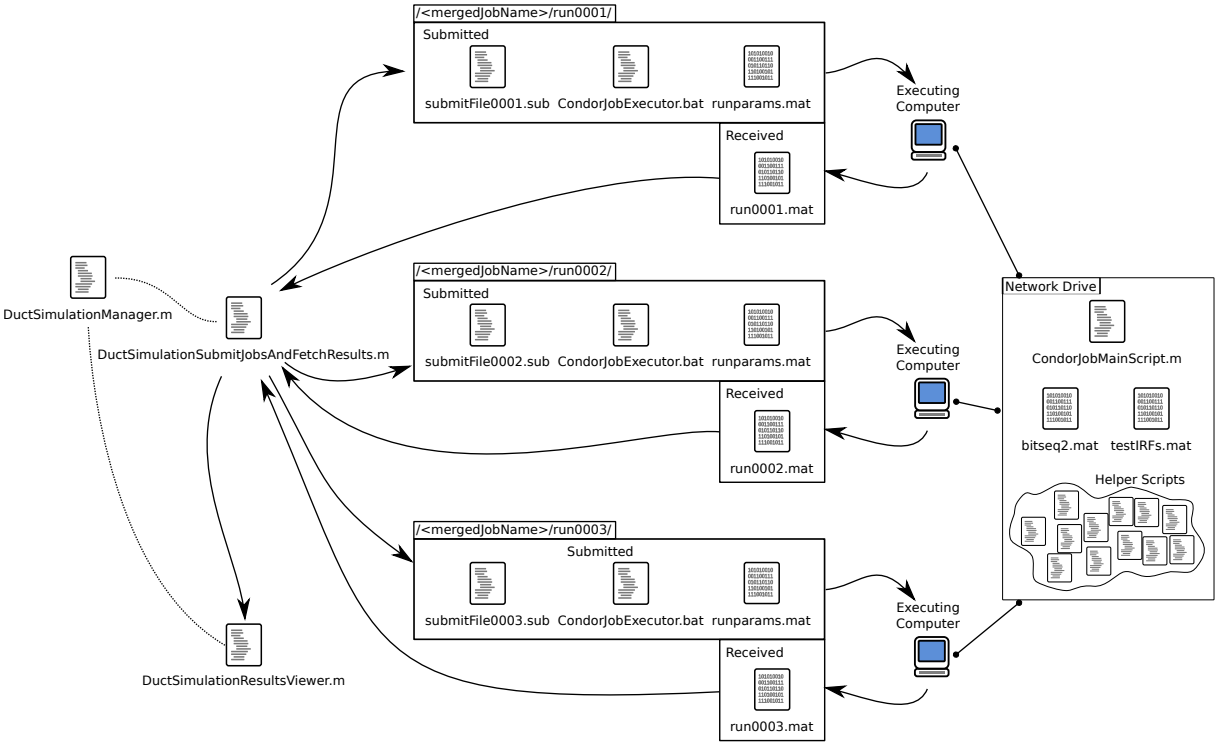


Figure 5.15: Schematic of the program execution.

3. Testing each inverse filter design with the communication system.

Each of these steps is described in greater detail in the following sections. It should be noted that schematic and the computer code show the use of fractional sampling and the RLS adaptive equalisers; however these were turned off during the main simulations as these required extra computational time that limited the turn-around whilst developing the simulation code.

5.3.3.2 Generation of the communication signal

The digital data sequence is taken from the remote network drive to ensure all experiments use the same signal. To transmit the digital data sequence over the channel, the sequence is required to be modulated to a suitable analog signal. This process is described in Section 2.2.4 and consists of mapping the digital sequence to a series of complex values using the Phase Shift Keying (PSK) pattern to create a complex sequence. A spectral shaping filter is designed and used to convert the complex sequence to a complex base-band signal used in the simulation.

5.3.3.3 Generation of the inverse filters

The channel IRFs used in the simulation are those recorded from the experiment discussed in Section 5.2. The channel IRFs are loaded from a file stored on the network drive. The channel IRFs are converted to base-band, and down-sampled to reduce the computations required to design the inverse filters (see Section 5.2.5), and corresponding receiver filters (if the filter design caters for a receiver filter). After the filters are designed, they are up-sampled for use in the testing of the inverse filter. The inverse filters were generated and tested for each regularisation parameter in the range provided in the file `runparams.mat`.

5.3.3.4 Testing of the inverse filter

The testing of the inverse filter consists of using the communication signal as the virtual source signal for the target receiver, whilst setting the signal for the non-target virtual source to zero. These signals are filtered by the inverse filter to generate the signals that become the base-band transmitter signals. The base-band transmitter signals are then filtered by the base-band channel IRFs to produce the base-band signals that would be expected at the receivers. The signals are then passed through the receiver filter, and then used with a symbol synchronisation process to determine the delay, and complex amplitude and phase. These synchronised results are used to sample the signal. The sampled signal is then passed through both non-adaptive and adaptive filters. The adaptive filters were applied using two sets of training sequences: no training sequence, or a sequence of 40 symbols.

Various values were measured throughout the simulation as indicated by the multimeter icon in Figure 5.16. The values recorded were:

- Peak value and power of the transmission signals.
- For a transmission to the target receiver: The power of the entire signal at the target receiver, the power of the signal before the portion of the signal containing the communication signal, and the power of the communication signal only.
- For a transmission to the non-target receiver: The power of the entire signal at the target receiver, and the power of the communication signal only.
- Power of the signal after the receiver filter for both the target and non-target receiver.
- Standard deviation of the sampled symbols without the use of a detector. (Two methods of measuring the standard deviation were performed, as discussed in the source-code.)

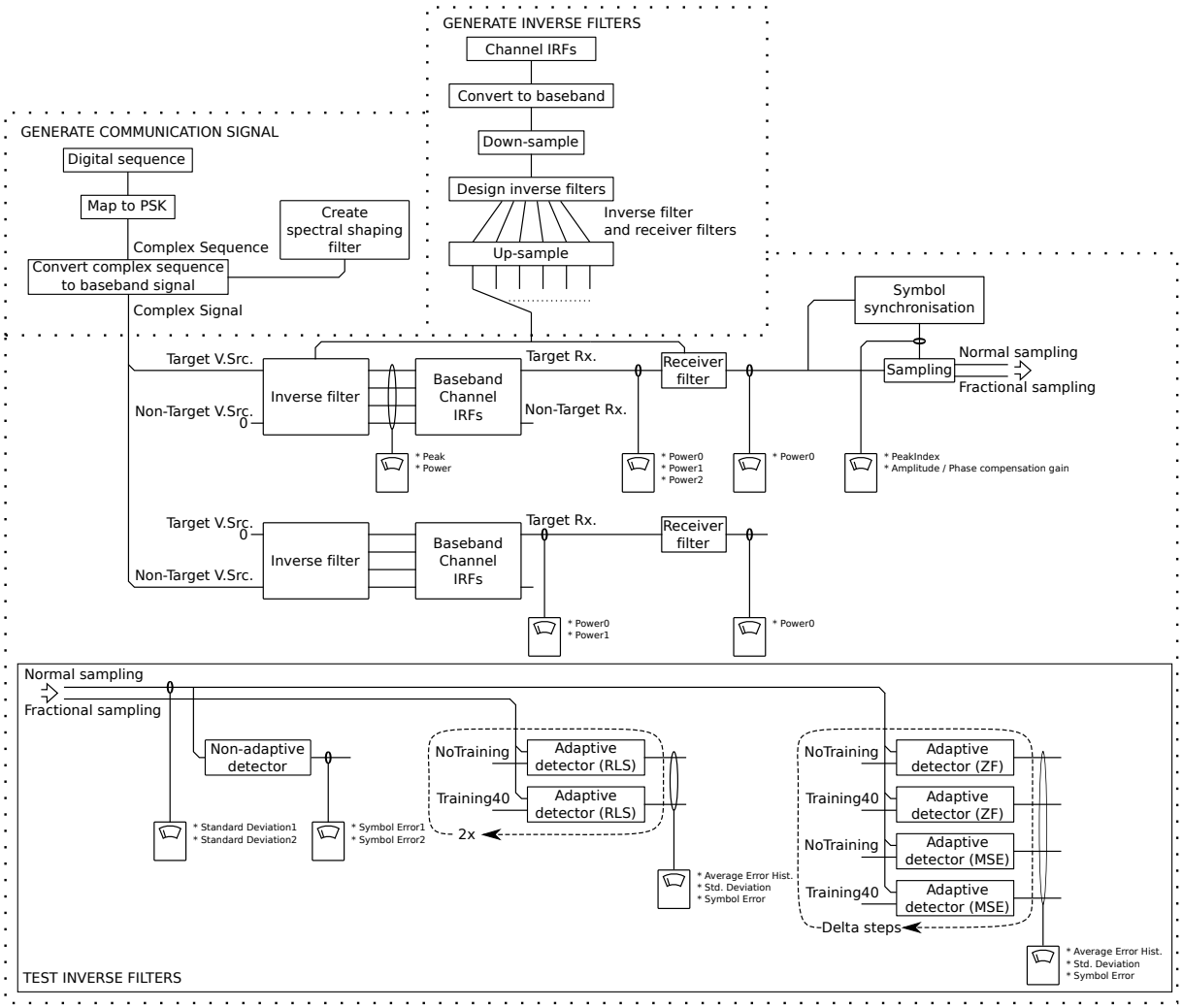


Figure 5.16: Schematic of the simulation executed on each computer.
Whilst the schematic and the computer code show the implementation of fractional sampling, and RLS adaptive equalisers, these were turned off during the main simulations due to computation limitations.

- The symbol error using a non-adaptive detector.
- The average symbol error at each iteration of the application of the adaptive filters.
- The standard deviation of the symbols for each of the adaptive filters used.
- The symbol error for each of the adaptive filters used.

5.3.4 Conclusions

In this section the programs used to perform a simulation of the experiment conducted in Section 5.2 have been presented. The purpose of this section was to describe the programs used to perform the simulation of the inverse filters. Whilst various changes and bug fixes were required during its development, the simulation proved to be a success. The results that were obtained from the simulation have been published in a journal paper [Dumuid et al., 2008], and are included in Chapter 6, which describes the inverse filter designs, and the results in greater detail.

6 Performance of Tikhonov regularised inverse filter design structures

The purpose of this chapter is to analyse the results of the simulation described in Chapter 5 that examined various Tikhonov inverse filter implementations over a range of scenarios and parameters. This chapter describes the theory behind the filter design structures implemented, and provides the results from the simulation, along with a discussion of the performance of the various filter structures examined.

The work presented in this chapter has previously been published in the journal paper entitled *"A comparison of filter design structures for multi-channel acoustic communication systems"*, by P. Dumuid, B. Cazzolato and A. Zander, published in the Journal of Acoustical Society of America Vol. 123 (1), pp. 174-185, Jan 2008.

6.1 Introduction

In Section 3.2 the application of channel compensation methods to communication systems was described. Stojanovic [2005] showed that time-reversal (TR) was severely limited in its performance when compared to a number of alternative designs. Cazzolato et al. [2001] demonstrated that inverse filtering outperformed time-reversal when using a Dirac impulse transmission in a simulated underwater environment. In this chapter, the research conducted in this thesis to extend the initial work by Cazzolato et al. [2001] is presented.

Different ways of implementing Tikhonov inverse filtering are investigated in conjunction with PSK communication using a simulation to examine the influence of a number of parameters. The performance of each of these implementations are compared against TR and a design presented by Stojanovic [2005]. As the filter design presented by Stojanovic is a theoretical design, the filter was modified slightly to make it practically realisable.

In this chapter, comparisons are made between the inverse filter designs based on how well they reduce the symbol errors for both a single channel

and multi-channel transmission. The inverse filter designs were compared via a simulation that used measurements obtained from the Impulse Responses (IRs) measured from a reverberant open-air waveguide that is described Section 5.2.

The results obtained from the simulation demonstrate that Tikhonov inverse filtering with appropriate regularisation performs better than time-reversal. The Stojanovic two-sided filter design is observed to outperform both time-reversal and Tikhonov inverse filters when used in a single channel scenario. However, the multi-channel implementation of the Tikhonov inverse filter is found to be the only filter able to perform multi-channel communication for the channel IRs investigated here.

6.2 The filter designs

6.2.1 Design classifications

Three filter classifications are presented as a means by which inverse filters may be designed for Multiple Input - Multiple Output (MIMO) systems. They are: *inverse by path* (SISO), *inverse by channel* (MISO) and *inverse by full MIMO.* These classifications are based on the filter structure shown in Figure 6.1. A set of sources, s_i, $i \in [1, S]$, are desired to be replicated at the receivers, r_i, $i \in [1, S]$, with minimal cross-talk through the use of a set of transmitters, t_j, $j \in [1, T]$. The channel through which the signals are transmitted is also modelled as a set of IRs, $c_{i,j}(t)$, $i \in [1, S]$ and $j \in [1, T]$ denote the receiver and transmitter respectively. In order to achieve the desired response between the source and the receiver a set of filters, $h_{j,i}$, are generated.

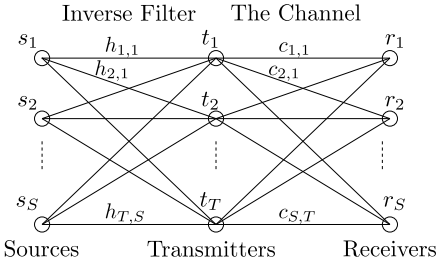


Figure 6.1: Schematic of filter connections.

The design of the inverse filters can be classified by which channel responses each inverse filter depends on. Figure 6.2a presents a classification that shall be called *inverse by path.* In this design, each sub-filter, $h_{j,i}(t)$, of the inverse filter is only dependent on a single channel response,

$c_{i,j}(t)$. The *inverse by path* classification thus encompasses the inverse filter designs that are multi-channel filters developed by combining multiple single channel systems. The general aim of filter designs developed according to the *inverse by path* classification is to find the inverse filter, $h_{j,i}$, such that $h_{j,i}(t) * c_{i,j}(t) \simeq \delta(t)$, where $*$ is the convolution operator, and $\delta(t)$ the Dirac delta function. If only the source, $s_i(t)$, is excited then the response at receiver, i, from the combination of all the individual systems is given by

$$r_i(t) = \left(\sum_{j=1}^{T} h_{j,i}(t) * c_{i,j}(t) \right) * s_i(t) \tag{6.1}$$

$$\simeq T\delta(t) * s_i(t), \tag{6.2}$$

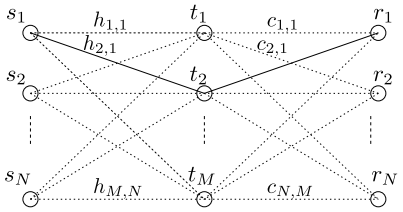
where the summation in Equation 6.1 is approximately $T\delta(t)$ in a multi-path environment as a result of the correlated peak at $t = 0$ due to the fact that $h_{j,i}(t) * c_{i,j}(t)$ should be uncorrelated for $t \neq 0$ and should thus average to zero. In other words, the T systems add coherently, increasing the gain T times at $t = 0$, whilst reducing the signal level away from $t = 0$. The cross-talk for the *inverse by path* design at receiver i can be determined by measuring the response at this receiver due to a signal that is transmitted to target a different receiver, $r_{i_{ct}}$, where the subscript ct stands for "cross-talk". The response is given by

$$r_i(t) = \left(\sum_{j=1}^{T} h_{j,i_{ct}}(t) * c_{i,j}(t) \right) * s_{i_{ct}}(t). \tag{6.3}$$

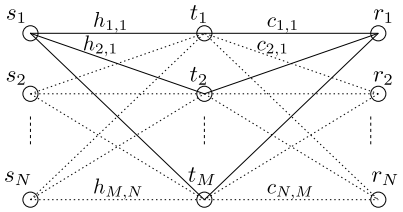
The cross-talk is observed to be dependent on the correlations between the filter, $h_{j,i_{ct}}(t)$, and the channel path, $c_{i,j}(t)$, for $j \in [1, T]$. If the channels are sufficiently uncorrelated then the term in the brackets should tend towards zero as the number of transmitters is increased. The time-reversal mirror is an example of an inverse filter designed according to the *inverse by path* design and its features of both focusing and temporal compression have been described in the literature extensively.

The second classification, called *inverse by channel* is shown in Figure 6.2b. This classification encompasses the filter designs where the sub-filters, $h_{j,i}(t)$, $j \in [1, T]$ are calculated together to replicate the response, $s_i(t)$, at receiver, r_i. This design is different from the *inverse by path* design since the filter, $h_{j,i}(t)$, is dependent on multiple responses, $c_{i,j}(t)$, $j \in [1, T]$. The cross-talk for inverse filters designed by the *inverse by channel* classification is dependent on the relationship between $h_{j,i_{ct}}(t)$ and $c_{i,j}(t)$ being uncorrelated in a similar manner to that for filters designed according to the *inverse by path* classification. The filters presented in Stojanovic [2005] are examples of inverse filters designed according to the *inverse by channel* classification.

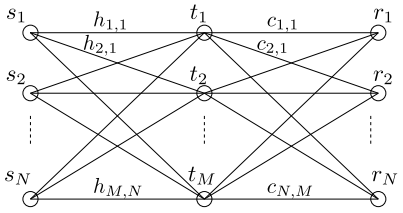
The last classification is called *inverse by full MIMO.* Filters designed according to this classification require that each sub-filter of the inverse filter is calculated based on all the channel responses. Inverse filters designed according to this classification attempt to achieve a MIMO transfer function between sources and receivers that has a desired response between the source and receiver, whilst also minimising the cross-talk between channels. Examples of this type of inverse filter design include those by Cazzolato et al. [2001], Montaldo et al. [2004], and Higley et al. [2006].



(a) inverse by path



(b) inverse by channel



(c) inverse by full MIMO

Figure 6.2: Schematic of filter design classifications. Solid lines denote transmission paths considered when developing filters. Dashed paths are additional paths which contribute to cross talk.

It should be noted that there are extensions to the design classifications discussed above, for example a number of the designs by Stojanovic [2005] include filters at both the transmitter and receiver.

6.2.2 The Tikhonov regularised inverse filter

The development of the Tikhonov regularised inverse filter is presented in Section 3.2.2. The fast Tikhonov inverse filter design using FFT was given

by (Equation 3.29)

$$\mathbf{H}(z) = \left(\mathbf{C}^{\mathrm{H}}(z^{-1})\mathbf{C}(z) + \kappa\mathbf{I}\right)^{-1}\mathbf{C}^{\mathrm{H}}(z^{-1})\mathbf{A}(z) \tag{6.4}$$

with the corresponding frequency domain equivalent (Equation 3.30)

$$\mathbf{H}(\omega) = \left(\mathbf{C}^{\mathrm{H}}(\omega)\mathbf{C}(\omega) + \kappa\mathbf{I}\right)^{-1}\mathbf{C}^{\mathrm{H}}(\omega)\mathbf{A}(\omega). \tag{6.5}$$

The Tikhonov inverse filter designs for each classification shall be defined given the inverse filter expressed as

$$\mathbf{H}(\omega) = \begin{bmatrix} H_{1,1}(\omega) & H_{1,2}(\omega) & \cdots & H_{1,J}(\omega) \\ H_{2,1}(\omega) & H_{2,2}(\omega) & \cdots & H_{2,J}(\omega) \\ \vdots & \vdots & \ddots & \vdots \\ H_{I,1}(\omega) & H_{I,2}(\omega) & \cdots & H_{I,J}(\omega) \end{bmatrix}, \tag{6.6}$$

the filter design according to *inverse by path* design is calculated by

$$\begin{aligned} H_{i,j}(\omega) &= \frac{C^{*}_{j,i}(\omega)}{C^{*}_{j,i}(\omega)C_{j,i}(\omega) + \kappa} \\ &= \left(|C_{j,i}(\omega)|^2 + \kappa\right)^{-1} C^{*}_{j,i}(\omega), \end{aligned} \tag{6.7}$$

the filter design according to *inverse by channel* is calculated by

$$\begin{aligned} \begin{bmatrix} H_{1,j}(\omega) \\ H_{2,j}(\omega) \\ \vdots \\ H_{N,j}(\omega) \end{bmatrix} = &\left(\begin{bmatrix} C^{*}_{j,1}(\omega) \\ C^{*}_{j,2}(\omega) \\ \vdots \\ C^{*}_{j,N}(\omega) \end{bmatrix} \begin{bmatrix} C_{j,1}(\omega) & C_{j,2}(\omega) & \cdots & C_{j,N}(\omega) \end{bmatrix} + \kappa\mathbf{I} \right)^{-1} \\ &\times \begin{bmatrix} C^{*}_{j,1}(\omega) \\ C^{*}_{j,2}(\omega) \\ \vdots \\ C^{*}_{j,N}(\omega) \end{bmatrix} \end{aligned} \tag{6.8}$$

and the filter design according to *inverse by full MIMO* is calculated by

$$\mathbf{H}(\omega) = \left(\mathbf{C}^{\mathrm{H}}(\omega)\mathbf{C}(\omega) + \kappa\mathbf{I}\right)^{-1}\mathbf{C}^{\mathrm{H}}(\omega). \tag{6.9}$$

In Chapter 4, it was shown that as κ tended towards infinity the resulting equation for the *inverse by full MIMO* filter, $\mathbf{H}(\omega)$, tended towards the time-reversal filter. Observing Equations 6.7 and 6.8, this property holds true for the Tikhonov inverse filter designs by *inverse by path* and *inverse by channel* classifications. Thus, as κ approaches infinity, each of these filtering classifications approaches the time-reversal filter design.

6.2.3 Regularisation of Stojanovic's two-sided filter for no inter-symbol interference

Stojanovic [2005] compared time-reversal with a set of optimal equalisation designs. These designs were developed for systems comprised of either multiple transmitters or multiple receivers, but not both. The Stojanovic filter design used in the simulation performed here is the "two-sided filter" that utilises multiple transmitters. Stojanovic [2005] proposed another filter that theoretically performed better by allowing ISI, however it was more difficult to implement and the performance improvement minimal and thus not used in the simulation presented here.

Using the notation presented in Figure 6.1, the two-sided filter design developed by Stojanovic [2005] between source, n and receiver, n is given by [Stojanovic, 2005, Eq. 19]

$$R_n(\omega) = K_n(\omega)\sqrt{X(\omega)}\gamma_n^{-1/4}(\omega) \tag{6.10}$$

$$H_{m,n}(\omega) = K_n^{-1}(\omega)\sqrt{X(\omega)}\gamma_n^{-3/4}(\omega)C_{n,m}^*(\omega) \tag{6.11}$$

where $R_n(\omega)$ is the filter at the receiver, $\gamma_n(\omega) = \sum_{m=1}^{M}\left|C_{n,m}^2(\omega)\right|$ is the composite channel power spectral density, $X(\omega)$ the Nyquist transfer function and

$$K_n(\omega) = \sqrt{\frac{E/\sigma_d^2}{\int_{-\infty}^{+\infty}\sqrt{S_w(\omega)}\frac{X(\omega)}{\sqrt{\gamma_n(\omega)}}d\omega}}S_w^{1/4}(\omega) \tag{6.12}$$

where E represents the transmission energy, σ_d^2 the average power of the data sequence, and $S_w(\omega)$ the power spectral density of the noise. This design can be extended to a MIMO design by designing the filters over all the $n = [1, N]$ source / receiver combinations.

The filter design by Stojanovic [2005] can be analysed by separating the equations into four different parts:

1. $K_n(\omega)$ - a filter that is related to the noise spectrum. If the noise is flat then $K_n(\omega)$ is constant with respect to frequency.

2. $X(\omega)$ - the Nyquist transfer function, being the raised cosine spectrum. The raised cosine spectrum is used as the desired total transfer function that has no ISI, and is commonly observed to be split between the transmitter and receiver (see Proakis [2001, Pg. 561]).

3. $\gamma_n^{-1/4}(\omega)$ and $\gamma_n^{-3/4}(\omega)$ - the inverse of the composite channel power spectrum. The composite channel power spectrum can be observed to be the total channel response when using time-reversal, and thus these filters combined are an inverse filter.

4. $C^*_{n,m}(\omega)$ - the time-reversal filter.

The two-sided filter given by Equations 6.10 and 6.11 can thus be described as a time-reversal filter (part 4) with inverse filters (part 3) fitted at both the transmitter and receiver to compensate for the summation of the time-reversals (through the inversion of γ_n) combined with a filter to compensate for the noise spectrum (part 1) and a filter to achieve the desired response (part 2) that is split between the source and receiver.

Stojanovic [2005] examined the theoretical performance of the two-sided filter. Such an examination does not consider the problems that occur when implementing the filters as time domain filters. In the work presented in this thesis, a small adjustment must be made to the design to account for non-causal wrap-around that could possibly occur when implementing the filter in the time-domain. Wrap-around may occur when converting the filters from the frequency domain into time domain if there are zeros in the composite channel power spectrum. To avoid wrap-around, a regularisation parameter, κ, can be added to the inversion of $\gamma_n(\omega)$, given in Equations 6.10 and 6.11, to produce a regularised filter design,

$$\begin{aligned} R_n(\omega) &= K_n(\omega)\sqrt{X(\omega)}\left(\gamma_n(\omega)+\kappa\right)^{-1/4} e^{-jT\omega/2} \\ H_{m,n}(\omega) &= K_n^{-1}(\omega)\sqrt{X(\omega)}\left(\gamma_n(\omega)+\kappa\right)^{-3/4} e^{-jT\omega/2} C^*_{n,m}(\omega) \end{aligned} \quad (6.13)$$

The term $e^{-jT\omega/2}$ is added to make the filter causal, where T is the duration of the FFT window.

6.3 Performance comparisons

6.3.1 Procedure

The impulse responses used in the simulation were measured from the laboratory experiment described in Section 5.2. The arrays were separated by an approximately 1 metre long open-air channel containing various objects to increase the reverberation. Reverberation was desired to emulate a difficult environment through which to transmit, such as an underwater acoustic environment. Figure 5.2 shows a photograph of the experimental setup. The quality of the speakers and microphones used in the experiment was rather low to reduce the cost of the experiment. The poor quality of the speakers and microphones described in Section 5.2 was considered to add an extra degree of difficulty for the inverse filters to compensate.

A number of parameters influenced the performance of the communication system including the carrier frequency, symbol rate and the regularisation parameter. Each filter was examined under a wide range of parameters in order to avoid the possibility that a certain set of conditions would favour

one particular inverse filter design over another. The inverse filters examined were:

1. Two-sided filter design by Stojanovic [2005]
2. Time-reversal filter
3. Tikhonov regularised filter formed by
    a) *inverse by path*
    b) *inverse by channel*
    c) *inverse by full MIMO*

To perform the simulation, a number of computers were controlled using Condor–a distributed computing software. Each computer calculated the system performance for a set of design parameters. A schematic of the simulation performed on each computer is shown in Figure 5.16. The parameters varied in the simulation were the number of transmitter elements, symbol rate, carrier frequency, regularisation parameter and filter type. A detailed description of the implementation of the simulation is given in Section 5.3. The job performed on each computer was as follows:

1. A digital sequence of 800 bits was used to generate 400 symbols using 4-PSK signal space. These symbols were used to create a base-band signal using the raised cosine spectral shaping filter.
2. The channel impulse responses (being sampled at a rate of 55 kHz) were converted to base-band and used to calculate each inverse filter.
3. The base-band signal from step 1 was convolved with each inverse filter to generate the signal that would be transmitted. The response at receiver 1 was examined under two conditions: firstly, for a transmission that is intended to be received at receiver 1, and secondly for a transmission that is intended to be received at receiver 2. The first condition was used to measure the quality of the transmission, and the second condition to measure the level of cross-talk.
4. The peak level and power of the transmission signals were measured and used to normalise all the measurements. In a real-time situation, the transmission signals would be normalised by an automatic gain control.
5. The transmission signals were convolved with the channel responses to generate the signals that would be received at the receiver array.

6. The power of each received signal was measured to determine the strength of the signal that would be received at the receiver.

7. The signal was synchronised with the first peak of the training sequence, sampled at the symbol rate and adjusted for amplitude and phase variation.

8. The sampled signal was passed into a non-adaptive and various adaptive detectors, from which the symbol error and standard deviation were measured.

It should be noted that whilst the schematic in Figure 5.16 and the computer code show the use of fractional sampling and the RLS adaptive equalisers, these were turned off during the main simulations as these required extra computational time, that limited the turn-around whilst developing the simulation code.

6.3.2 Sensitivity to noise

In the simulations, the environmental noise was omitted from the channel model so that the symbol errors would be the result of the inter-symbol interference (ISI) alone. The measured data can be post-processed to obtain a perspective of how sensitive the inverse filters are to environmental noise, and channel response mis-match errors.

If the symbol error resulting from ISI is distributed according to a Gaussian distribution, then the expected number of symbol errors when channel noise is included can be estimated by combining the standard deviation of the environmental noise and the standard deviation of the symbols resulting from ISI. This estimate is only possible provided adaptive filters are not used. If an adaptive filter were to be included, then the addition of environmental noise can actually be equivalent to adding regularisation (see Equation 10.2-33 in Proakis [2001]).

Whilst ISI cannot be assumed to be Gaussian, if it is smaller than the noise, then the error from estimating it as a Gaussian distribution is small [Shimbo and Celebiler, 1971]. The relationship between the probability of error for Gaussian interference having a standard deviation of σ for a PSK modulation scheme is given by [Proakis, 2001]

$$P_e = 1 - \left(Q(-\frac{1}{\sigma}) \right)^2 \tag{6.14}$$

where $Q(x) = \frac{1}{\sqrt{2\pi}} \int_{t=x}^{\infty} e^{-\frac{t^2}{2}} dt$ is the Matlab function, `qfunc`.

A scatter plot of the standard deviation versus the symbol error for the measurements is presented in Figure 6.3 along with the curve of the expected

symbol error obtained using Equation 6.14. The correlation between the measured data and the curve demonstrates that it is appropriate to estimate the inter-symbol interference as Gaussian. The graph of $400P_e$ approaches a limit of 300 symbol errors. This limit results because the modulation scheme (being 4-PSK) consists of four symbols, each with equal probability. The probability of a correct symbol is thus $\frac{1}{4}$, conversely the probability of an erroneous symbol, $P_e = \frac{3}{4}$, leading to $400P_e = 300$.

Using the relationship between the standard deviation and symbol error, it is possible to estimate:

- The ratio, R_0, by which the measured standard deviation could increase before obtaining a desired probability of error, P_i,

$$R_0 = \frac{\sigma_{P_i}}{\sigma_s} \tag{6.15}$$

- The root-mean-square environmental noise, n_0, that results in a desired probability of error, P_i,

$$n_0 = a_{rms}\sqrt{\sigma_{P_i}^2 - \sigma_s^2} \tag{6.16}$$

- The probability of error, $P_{e,CT}$, that would be expected as a result of the cross-talk

$$P_{e,CT} = 1 - \left(Q\left(\frac{-1}{\sqrt{\sigma_s^2 + \left(\frac{c_{rms}}{a_{rms}}\right)^2}} \right) \right)^2 \tag{6.17}$$

- The ratio, R_{CT}, by which the measured standard deviation could increase before obtaining a desired probability of error, P_i, when the cross-talk noise is included

$$R_{CT} = \frac{\sigma_{P_i}}{\sqrt{\sigma_s^2 + \left(\frac{c_{rms}}{a_{rms}}\right)^2}} \tag{6.18}$$

- The root-mean-square environmental noise, n_{CT}, that results in a desired probability of error, P_i, when the cross-talk noise is included

$$n_{CT} = a_{rms}\sqrt{\sigma_{P_i}^2 - \sigma_s^2 - \left(\frac{c_{rms}}{a_{rms}}\right)^2} \tag{6.19}$$

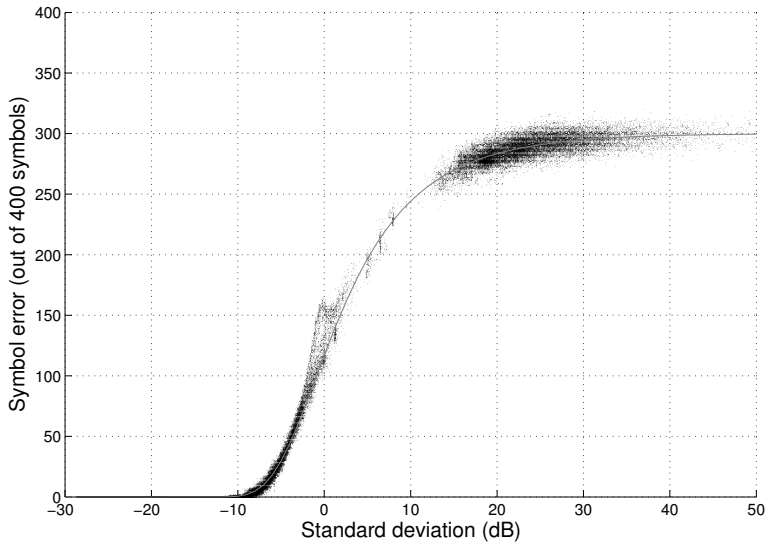


Figure 6.3: Scatter of standard deviation versus symbol error for all filter designs. The light curve shows the expected average for a Gaussian distributed symbol spread.

In each of these equations, σ_s is the standard deviation of the symbols, σ_{P_i} the standard deviation required to get a probability of error, P_i, c_{rms} is the measured root-mean-square measure of the cross-talk signal, and a_{rms} the root-mean-square measure of the received signal.

The ratio by which the standard deviation must increase to achieve a specific probability of error shows the sensitivity that the inverse filter design has to noise resulting from channel estimation error, cross-talk or environmental noise. The environmental noise required to achieve a probability of error differs from the ratio because the influence of environmental noise is dependent on the magnitude of the received signal. If the received signal is very large, then the environmental noise has a small impact on the standard deviation, however if the received signal is small, a small amount of environmental noise will have a significant impact on the standard deviation of the symbols.

6.4 Results

The experiment that was simulated consisted of utilising all six, three or two speakers of the transmitter array to transmit to two adjacent microphones at the receiver array. Two adjacent microphones were chosen to increase the cross-talk between the microphones to demonstrate the filter design performance. The average frequency responses between the six transmitters and each of the two receivers are shown in Figure 6.4. Receivers 1 and 2 are observed

to have a fairly similar response. Arbitrarily taking -25 dB as an operational level, the channel was considered operable between approximately 1.5 kHz and 20 kHz, despite the response fluctuating significantly throughout this range.

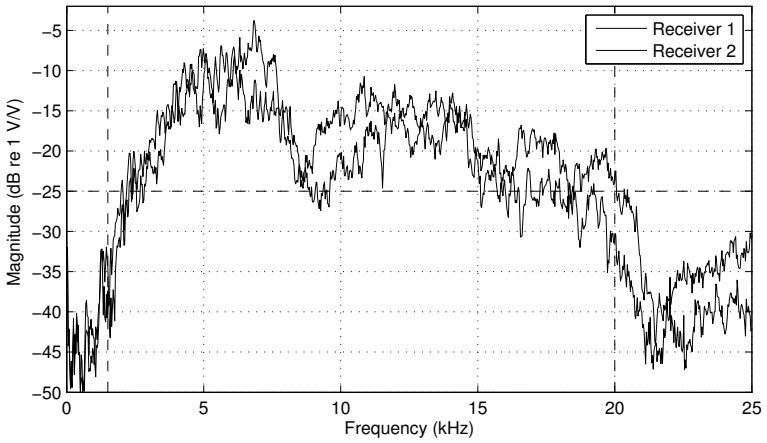


Figure 6.4: Average frequency responses between all 6 transmitters and receivers 1 and 2. The vertical dashed lines show the region in which the simulations occurred, and the horizontal dashed line indicates the chosen operational level.

Figures 6.5 to 6.19 show the results obtained from the simulations for each combination of transmitters, symbol rate, carrier frequency, regularisation parameter and filter design. The ranges over which the simulation parameters were varied are presented in Table 6.1. Each figure presents the results that are grouped according to the combination of transmitters. Each group contains a set of sub-plots placed in a grid, where the data rate varies horizontally between 2280, 5700, and 11400 baud, (being 25, 10, and 5 samples / symbol respectively) and the type of inverse filter used varies vertically. Each subplot shows the results for a range of regularisation values (ordinate) and carrier frequencies (abscissa). The results for no regularisation ($\kappa = 0$) are shown in the top row of pixels of each plot above the dashed line, whilst the remaining positions range logarithmically from $\kappa = 10^{-20}$ to 10^{0}. Since time-reversal does not have a regularisation parameter, the height of this subplot has accordingly been reduced. The regularisation parameter, κ, was normalised against the peak value of the frequency response of the channel, and results concerning powers at the receiver have been normalised against the total transmission power. The upper limit of 10^{0} for the regularisation parameter, κ, was chosen to significantly exceed the largest singular value of the channel responses. This value resulted in the filter effectively being a time-reversal filter. In some instances, results could not be

obtained because either the carrier frequencies were below half the symbol rate (Nyquist limit) or the received signal could not be synchronised to determine the desired signals. The regions where the results are not available have been masked with grey.

Table 6.1: Range of simulation parameters

Transmitter elements used	[1 2 3 4 5 6], [1 2 3] and [1 3]
Regularisation parameter, κ	$0, 10^{-20}, 10^{-19.5}, \cdots, 10^{0}$
Carrier frequencies	1.5 kHz , $\cdots$, 20 kHz (665 bins)
Data rate	2280, 5700 and 11400 baud
Filter design structure	Stojanovic two-sided Tikhonov inverse *by path* Tikhonov inverse *by channel* Tikhonov inverse *by full MIMO* time-reversal

General observations Several features can be observed from all the results presented in Figures 6.5 to 6.19. The first observation is that as the regularisation parameter is increased, the results tend to become similar to that of the time-reversal filter. This observation confirms the theoretical prediction presented in Chapter 4 that the inverse filters tend towards the time-reversal filter as κ tends towards infinity. A second observation is that as the regularisation parameter approaches zero, some of the filters become unstable. In such situations the signal that the receiver is required to process is severely corrupted. The regions at which this occurs can generally be observed to appear speckled or greyed out altogether. A third observation is that as the data rate is increased, the results appear smoother between carrier frequencies (provided the inverse filter is stable). This can be explained from the bandwidth occupied by the signal. At the low data rate of 2280 baud, the bandwidth occupied is ± 1.14 kHz (see Section 2.2.3), thus an increase of 2.28 kHz results in the signal operating in entirely different bandwidth space; however for the higher data rate of 11400 baud the bandwidth occupied is ± 5.7 kHz, and a shift of 2.28 kHz results in the signal occupying 80% of the previous frequency range, and thus is more likely to have similar performance.

Transmitter power Figure 6.5 shows the transmitter power prior to the application of any normalisation. It should be noted that an automatic gain control (AGC) would generally compensate for the variations observed in these plots, however it is of interest to observe the signal levels resulting from the filter designs. From the figures it can be observed that the signal

level is greater when $\kappa > 10^{-7}$, and the Stojanovic two-sided filter design generally has the greatest signal level. As the data rate is increased, the signal level is observed to diminish for all the filter designs.

Receiver power Figure 6.6 shows the power of the signal received at the target receiver. There appears to be only a minor variation of the power received with respect to the number of elements used at the transmitter, although reducing the number of elements generally increases the received power slightly. The increase in power could be attributed to the reduction of destructive interference.

The power received is generally greatest for the Stojanovic two-sided filter design whilst the Tikhonov *inverse by path* filter design is observed to have the least. Whilst the Stojanovic filter design is the least influenced by the regularisation parameter, as κ goes above 10^{-3}, all the filters appear to produce similar power and generally greater than for $\kappa < 10^{-3}$.

The high signal power received for the Stojanovic two-sided filter design can be understood by examining the structure of the filter. The structure consists of matched filters applied to each receiver followed by a summer that combines the outputs of these filters. The combined signal is passed through an inverse filter (shared between the source and receiver) that compensates for the signal, being the sum of these responses. The matched filter can be observed to maximise the signal-to-noise ratio, and the summation to result in an averaging that causes the frequency response to be smoother and less likely to have deep nulls, and thus should be easier to invert. On the other hand, the Tikhonov inverse filter is applied to compensate for responses that are less smooth, and more effort is used to smooth out the response, resulting in less received energy but of higher quality.

Amplitude of the sampled signal The average symbol amplitude of the synchronised and sampled signal is shown in Figure 6.7. Whilst this plot is related to the received power presented in Figure 6.6, the relationship is dependent on the performance of the inverse filter. If the inverse filter is functioning correctly, then the amplitude of the received signal at the sampling instance should be a fixed amplitude that would be related to the receiver power. If the inverse filter was not functioning correctly, the received signal level for each sampling instance would vary and thus the average amplitude would be smaller regardless of the receiver power. However, a high amplitude does not directly correlate to good inverse filter performance in terms of inter-symbol interference. It is well known that the time-reversal filter (or matched filter) achieves the highest signal-to-noise ratio (which would likewise result in a large average sampling amplitude), however this is at the expense of inter-symbol interference.

Figure 6.7 shows areas that are quite similar in magnitude to the received power, however there are also areas where the amplitude is greatly reduced and also appears noisy–typical of an unstable inverse filter. It is of particular interest to note that despite the high received power from the Stojanovic two-sided filter design at 5700 baud, the symbol amplitude is actually increased by using higher values of regularisation.

Ratio of receiver power to cross-talk power Figure 6.8 shows the ratio of the receiver power to the cross-talk power (being the power of the signal at the receiver for a transmission targeted at the cross-talk receiver). The Tikhonov inverse filter designed by *full MIMO* has the largest signal to cross-talk noise ratio, ranging between 30 and 35 dB for the optimal carrier and regularisation parameter values for the six transmitter configuration, and between 10 and 15 dB for the two transmitter configuration. This performance is far above that of all the other filter designs, typically having a ratio between -5 and 10 dB, and greater for higher regularisation values, where all the filters tend towards being a TR filter, as also shown in the figure. The Stojanovic two-sided filter is shown to consistently have a slightly larger ratio than the Tikhonov *inverse by path* and *inverse by channel* filters.

The range of regularisation parameter that provides a good signal to cross-talk noise ratio for the full Tikhonov inverse filter is reduced when the data rate is increased or a smaller number of transmitters are used. This can be attributed to the fact that when the data rate is increased, the bandwidth of the signal is also increased resulting in the inverse filter being required to place more effort on compensating for the channel fluctuations. When the number of transmitters is reduced, the inverse filters have fewer channels to compensate zeros in the transmission path, and resulting in poor composite transfer functions.

Cazzolato et al. [2001] showed that the focal region of the Tikhonov *inverse by full MIMO* filter was smaller than that for time-reversal. Thus, the cross-talk resulting for time-reversal could be reduced by an increase in the distance between the receiver elements. However, increasing the distance between the receivers would also improve the performance of the *inverse by full MIMO* filter as less effort would be required to eliminate the cross-talk.

Symbol error The symbol error measured after the received signal was synchronised and amplitude / phase adjusted is shown in Figure 6.9. The areas where low symbol error is observed denote the set of parameters for which the combination of the inverse filter and channel response result in a received signal from which the symbols could be detected. To obtain low symbol errors, the inverse filter design is required to result in a stable set of filters.

The Stojanovic two-sided filter has very low symbol error without the need for regularisation for data rates of 2280 and 5700 baud regardless of the number of transmitters used. However at the higher data rate of 11400 baud, the filter design requires a regularisation of between 10^{-11} and 10^{-4}. The Tikhonov *inverse by path* filter design can achieve zero symbol error without any regularisation for some carrier frequencies at low data rates, however the addition of regularisation of around 10^{-4} allows the inverse filter to achieve low symbol rates for all carrier frequencies and all transmitter arrangements. The Tikhonov filters designed by both *inverse by channel* and *inverse by full MIMO* require that regularisation always be used to achieve low symbol errors. All the inverse filters have symbol errors as the regularisation increases above 10^{-2}, and time-reversal is observed to have the worst performance of all the filter designs.

The range of suitable regularisation reduces with either an increase in the data rate or a decrease in the number of transmitters. The Tikhonov *inverse by full MIMO* filter design consistently has the smallest range of suitable regularisation values; whilst the Stojanovic two-sided filter has the largest.

The reason the Stojanovic two-sided filter requires little or no regularisation can be attributed to the fact that the inversion is performed on a spectral averaged response of all the channels. In order for a zero to occur within this average, all of the IRs must share a common zero. As the number of transmitters is increased, the probability of a common zero is reduced. At 11400 baud, regularisation was required for the Stojanovic two-sided filter to achieve a low number of symbol errors. Regularisation was found to be required for this symbol rate because a common zero existed in the frequency responses of the channels. This common zero was the result of the filtering performed in the band-pass to base-band conversion.

Sensitivity of the system to noise In Section 6.3.2 it was described how the standard deviation of the sampled symbols could be related to the symbol error and used to obtain an estimate of the sensitivity of the inverse filter to both environmental noise and channel estimation error. To validate the ability of using the standard deviation to determine the symbol error, the symbol error for 400 symbols calculated from the standard deviation is shown in Figure 6.10. Comparing the estimated symbol error with the actual symbol error shown in Figure 6.9, there appears to be reasonably good agreement, particularly around the edges of the regions having zero symbol errors. The accuracy of the boundaries of the zero symbol error are of particular interest as this is the value of standard deviation used to determine the margin by which the noise could increase.

The increase in the standard deviation before attaining a 1 in 400 chance of error is shown in Figure 6.11. The Stojanovic two-sided filter is observed to have the greatest margin. However, if the regularisation parameter is ap-

propriately chosen for either the *inverse by channel* or *inverse by full MIMO* Tikhonov inverse filter a similar level of performance is achieved. The *inverse by path* design is observed to perform the worst, whilst the *inverse by channel* filter is the most resilient to noise. This can be explained by the fact that the *inverse by path* filter uses most of the energy to compensate for nulls in each channel rather than for a combination of channels as per the *inverse by channel* or *inverse by full MIMO* designs. The *inverse by full MIMO* filter performs worse than the *inverse by channel* design since part of the effort in the filter design is used to reduce the cross-talk to the other receivers.

Figure 6.12 shows the noise level required in the channel to cause the standard deviation to increase to the point of achieving a probability of error, $P_e = \frac{1}{400}$. The noise level required to achieve this error is dependent on both the increase of standard deviation required to reach $P_e = \frac{1}{400}$, shown in Figure 6.11, and the magnitude of the received symbols shown in Figure 6.7. Comparing these figures, it is evident that provided the margin for the standard deviation of the received signals is above 2 dB, the noise margin is primarily related to the symbol amplitude.

The Stojanovic two-sided filter has the largest margin for the noise to increase at the higher data rates, however with an appropriate choice of regularisation parameters, for the correct carrier frequency, similar performance can be obtained from all the receivers. As the number of receivers is reduced, the performance of all the filters is slightly reduced. In particular, the range of regularisation values for which the Tikhonov inverse filter produces the near optimal results is reduced. The Stojanovic two-sided filter clearly performs the best with regard to sensitivity to noise.

Performance prediction with cross-talk In Section 6.3.2 it was shown that the standard deviation of the sampled symbols could be related to the symbol error and used to obtain an estimate of the sensitivity of the inverse filter to environmental noise, channel estimation error, and cross-talk interference. In this section, the results for the influence of cross-talk are presented. Figure 6.13 shows the estimated symbol error, Figure 6.14 the increase in standard deviation required to obtain a probability of error, $P_e = \frac{1}{400}$, and Figure 6.15 the required level of noise to obtain a probability of error, $P_e = \frac{1}{400}$, when cross-talk is considered.

The estimated symbol error presented in Figure 6.13 shows that the Tikhonov *inverse by full MIMO* filter design provides the greatest performance having obtained no symbols from cross-talk when appropriate regularisation is chosen, regardless of the carrier frequency or number of transmitters used. However the range of regularisation values over which the regularisation provides good results is reduced when the number of transmitters is reduced or the data rate is increased. Whilst the other inverse filter designs show certain amounts of error, many could be made to function through the

use of an adaptive equaliser. The Stojanovic two-sided filter has a low symbol error and the performance improves with an increase in data rate, but decreases when reducing the number of transmitters.

The increase in standard deviation or noise required to obtain a probability of error, $P_e = \frac{1}{400}$, shown in Figures 6.14 and Figure 6.15 demonstrate that the Tikhonov *inverse by full MIMO* is the only filter that can safely function in a multi-channel system for most carrier frequencies, data rates, and number of transmitters. The *inverse by channel* appears to be able to function at an appropriate error rate, however the range of carrier frequencies over which the error rate can be achieved is very small, and the margins for noise fluctuations extremely low. However, at high data rates or when using few transmitters, the margins for performing MIMO communication are greatly degraded. Whilst it was mentioned in the previous paragraph that adaptive filters could be used to achieve a functional multi-channel system using the other filter designs, the ratio of the target signal to cross-talk signal power presented in Figure 6.8 demonstrates that the Tikhonov *inverse by full MIMO* has the smallest sensitivity to the cross-talk noise and is observed to operate best when the regularisation is around 10^{-3} for lower carrier and 10^{-4} for higher carrier frequencies. At low symbol rates, the range of regularisation producing low symbol error is large, however for faster data rates, the range is much smaller. This can be attributed to the bandwidth that the signal occupies. When the data rate is increased, the bandwidth is increased, and the number of zeros likely to be within the bandwidth increases, resulting in a reduction in the range of regularisation values.

Symbol error for the adaptive equalisers A number of adaptive filters were implemented at the receiver having both feed-forward and feed-back taps that spanned 18 ms (half the length of the channel impulse responses). The responses were passed through the adaptive filters three times with a decrease in the step size to obtain the best possible adaption. The taps resulting from each step were used at the next iteration unless the average error increased. If the error increased the adaptive equaliser taps were reset to the state they were before the current iteration. The results from using the zero forcing and least-mean-square (LMS) adaptive filters, with and without a training sequence of 40 symbols, are shown in Figures 6.16 to 6.19. Comparing the symbol error to that without any adaptive filtering, as shown in Figure 6.9, the adaptive equalisers have been able to reduce the error, provided the error is not too large. Of particular note, the adaptive filter has enabled the time-reversal to operate over a much wider range of frequencies at the speed of 2280 symbols / second.

6.5 Conclusion

In this chapter, three classifications of channel filter design have been discussed; *inverse by path*, *inverse by channel* and *inverse by full MIMO*. Tikhonov regularised inverse filters were implemented according to these classifications and compared with the time-reversal filter and the two-sided filter design proposed by Stojanovic [2005]. Whilst Stojanovic [2005] presented theoretical results, the filter was shown to require a regularisation parameter in order to be practically implementable. The Stojanovic two-sided filter outperforms the Tikhonov regularised inverse filter designs when communicating over a single channel. The *inverse by path* and *inverse by channel* performed particularly poorly when compared to the Stojanovic two-sided filter, whilst the *inverse by full MIMO* design was found to have only slightly reduced performance. Whilst the Stojanovic two-sided filter and *inverse by path*, and *inverse by channel* Tikhonov inverse filter designs are not designed for MIMO communication; they were however found to reduce the cross-talk. However, the performance of these designs for MIMO communication was found to be poor when compared to the *inverse by full MIMO* filter. The *inverse by full MIMO* was found to provide 20 dB less cross-talk at the expense of around 2 dB loss in signal strength when compared to the Stojanovic two-sided filter. In this scenario, the *inverse by MIMO* Tikhonov filter design was the only design that was able to be used to transmit multiple transmission streams.

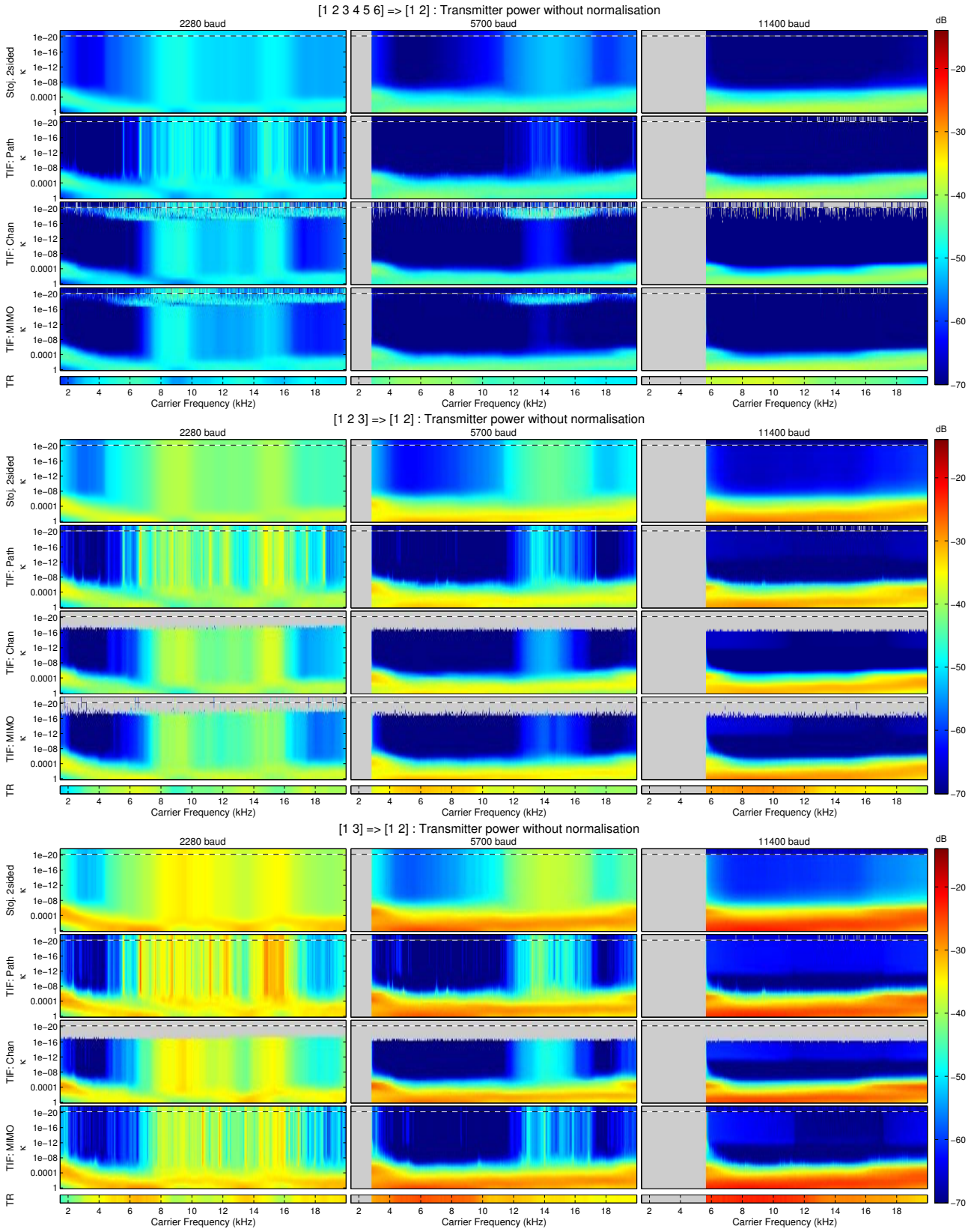


Figure 6.5: Transmitter power for the parameter ranges presented in Table 6.1. Results for $\kappa = 0$ are presented in the plots in the top row of pixels above the dashed line.

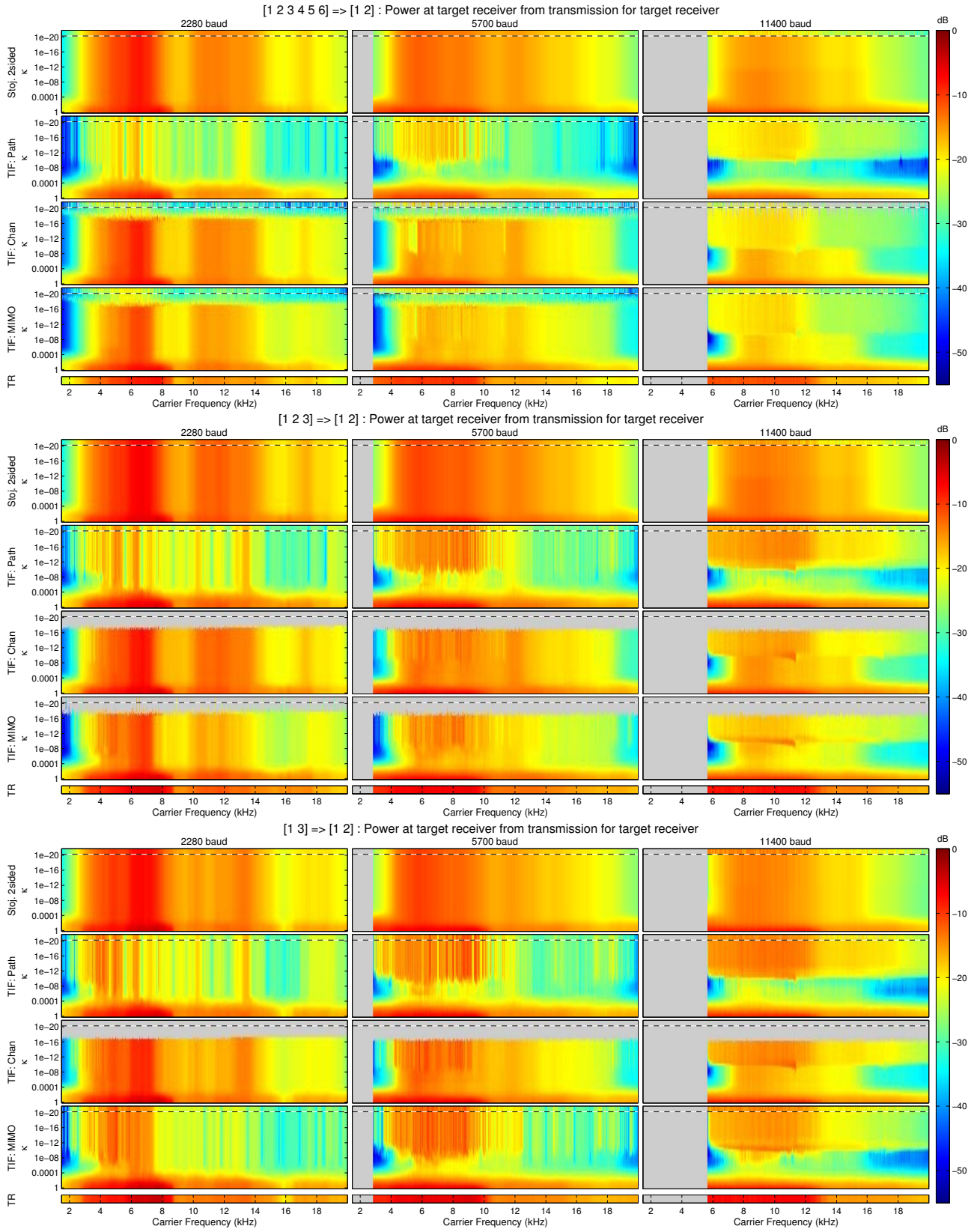


Figure 6.6: Power at the target receiver for the parameter ranges presented in Table 6.1. Results for $\kappa = 0$ are presented in the plots in the top row of pixels above the dashed line.

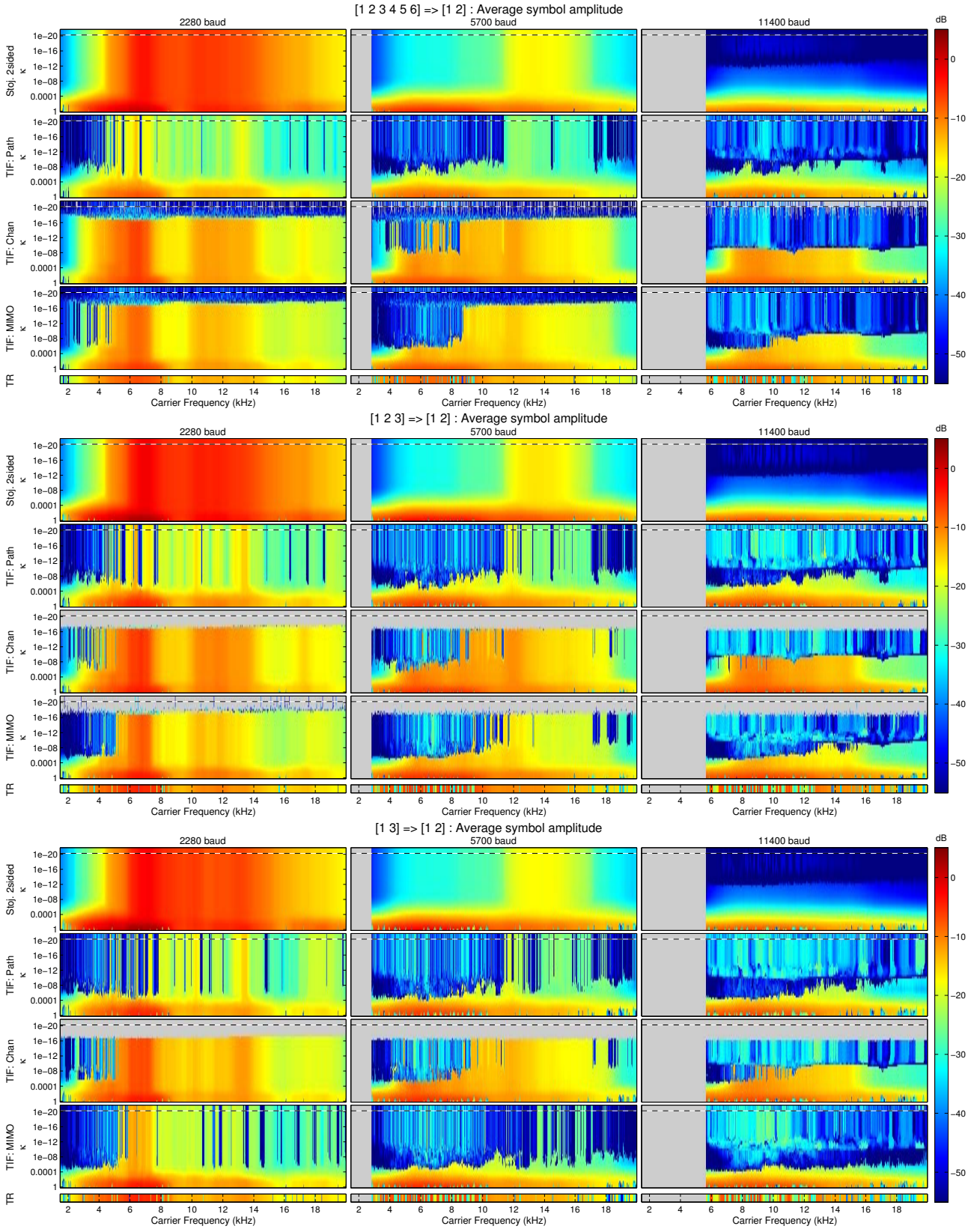


Figure 6.7: Average amplitude of sampled signal prior to compensation of the phase / amplitude for the parameter ranges presented in Table 6.1. Results for $\kappa = 0$ are presented in the plots in the top row of pixels above the dashed line.

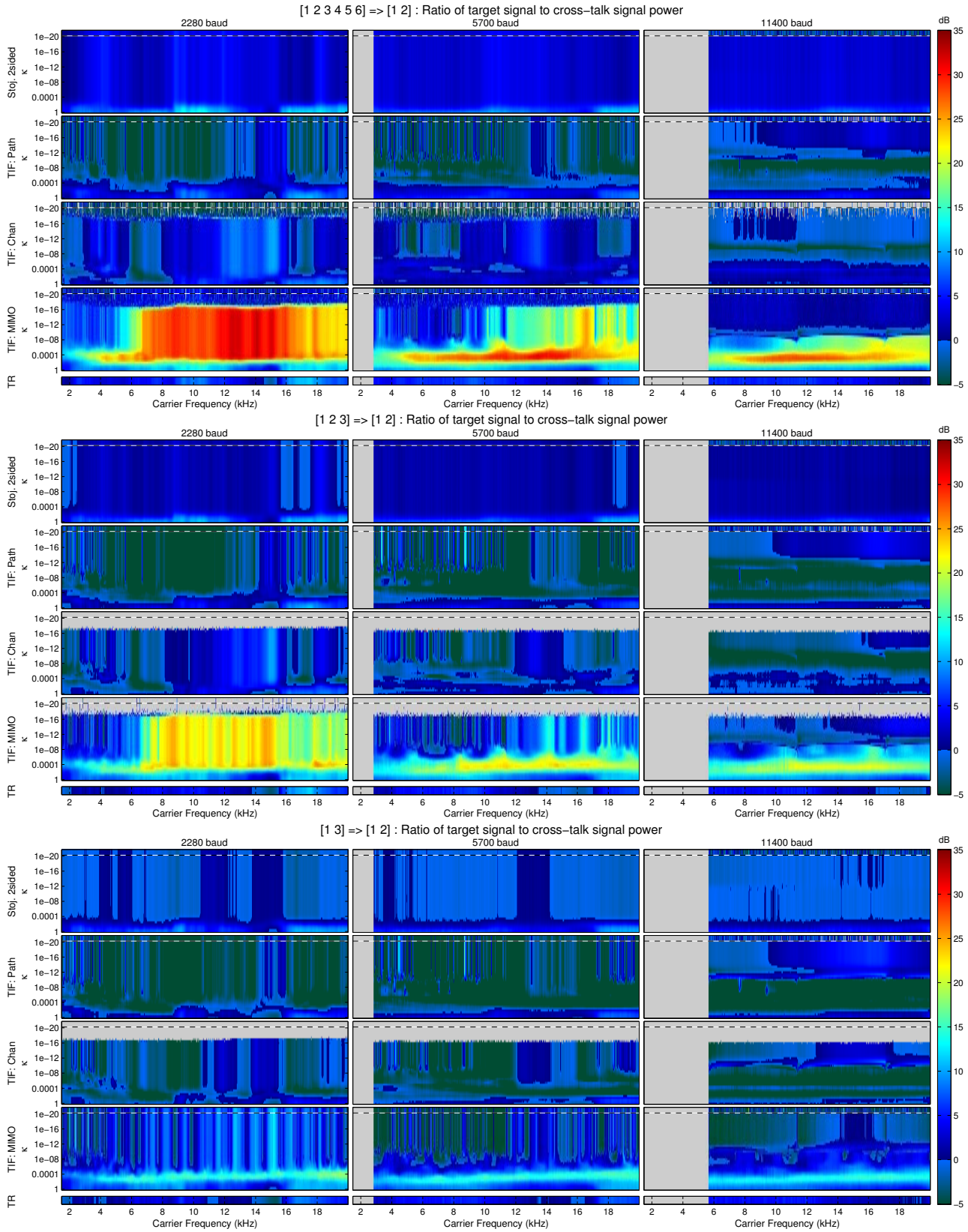


Figure 6.8: Ratio of the receiver power to the cross-talk power for the parameter ranges presented in Table 6.1. Results for $\kappa = 0$ are presented in the plots in the top row of pixels above the dashed line.

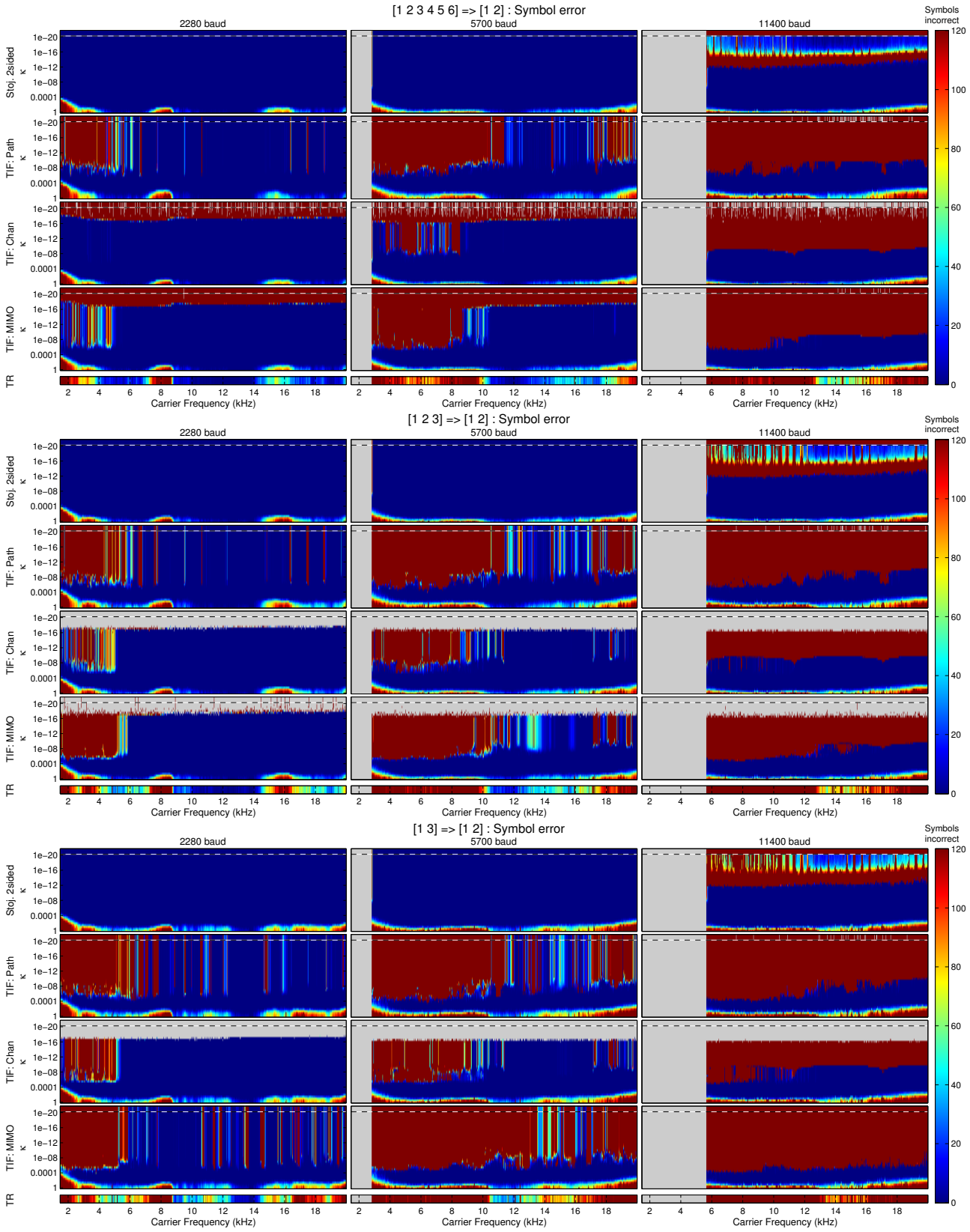


Figure 6.9: Symbol error without any adaptive filters for the parameter ranges presented in Table 6.1. Results for $\kappa = 0$ are presented in the plots in the top row of pixels above the dashed line.

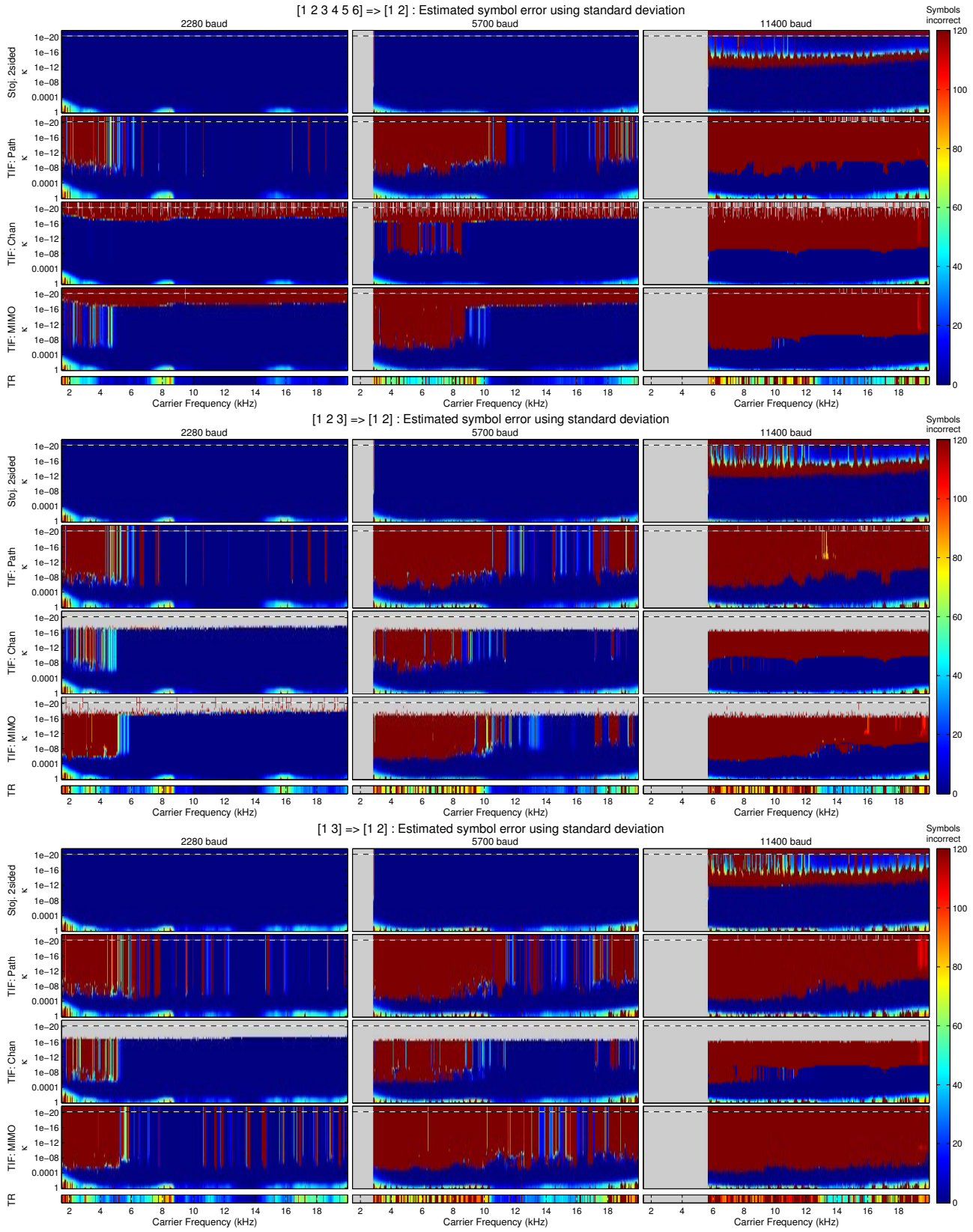


Figure 6.10: Estimate of symbol error derived from the standard deviation for the parameter ranges presented in Table 6.1. Results for $\kappa = 0$ are presented in the plots in the top row of pixels above the dashed line.

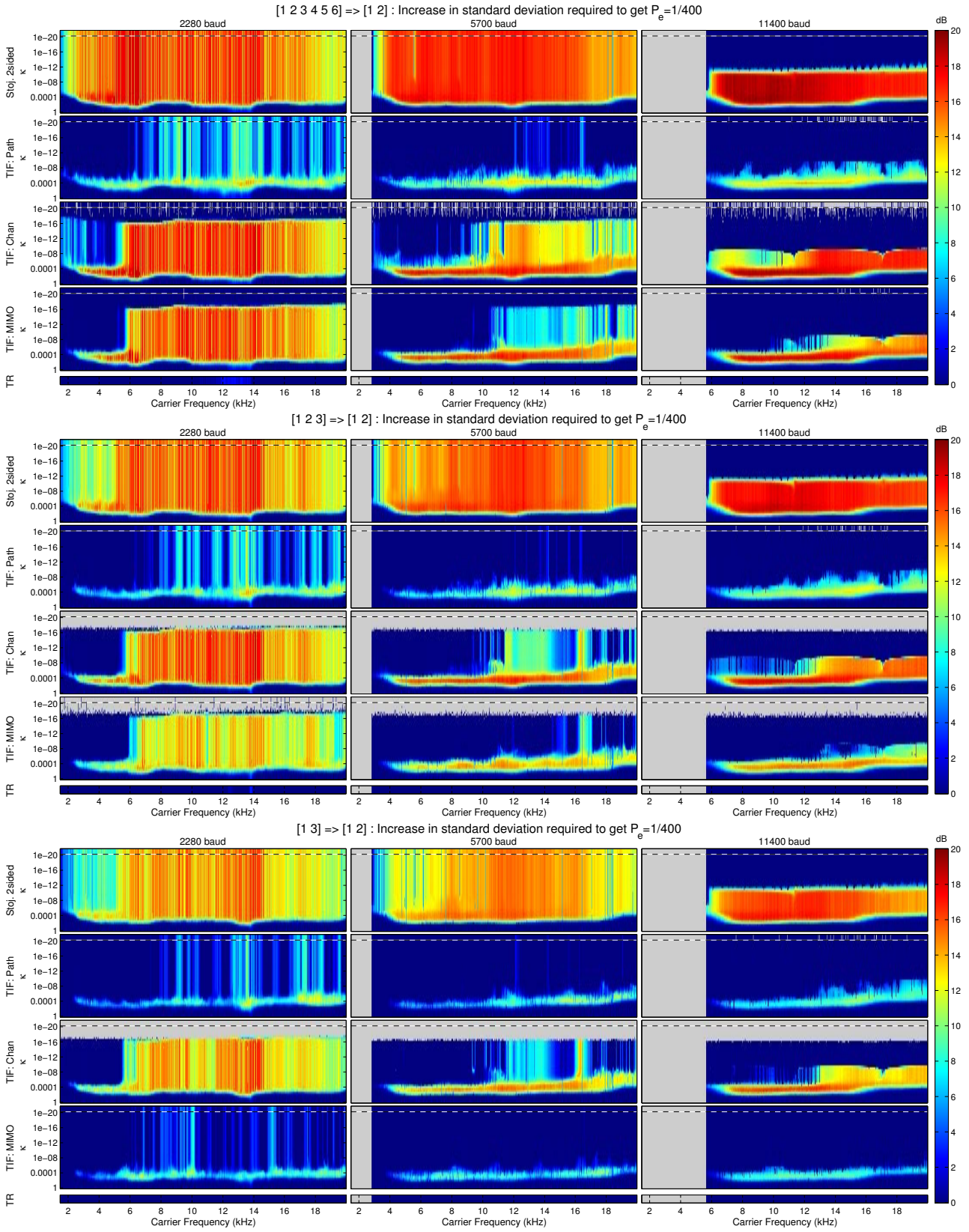


Figure 6.11: Increase in standard deviation required to achieve an error rate of 1 in 400 for the parameter ranges presented in Table 6.1. Results for $\kappa = 0$ are presented in the plots in the top row of pixels above the dashed line.

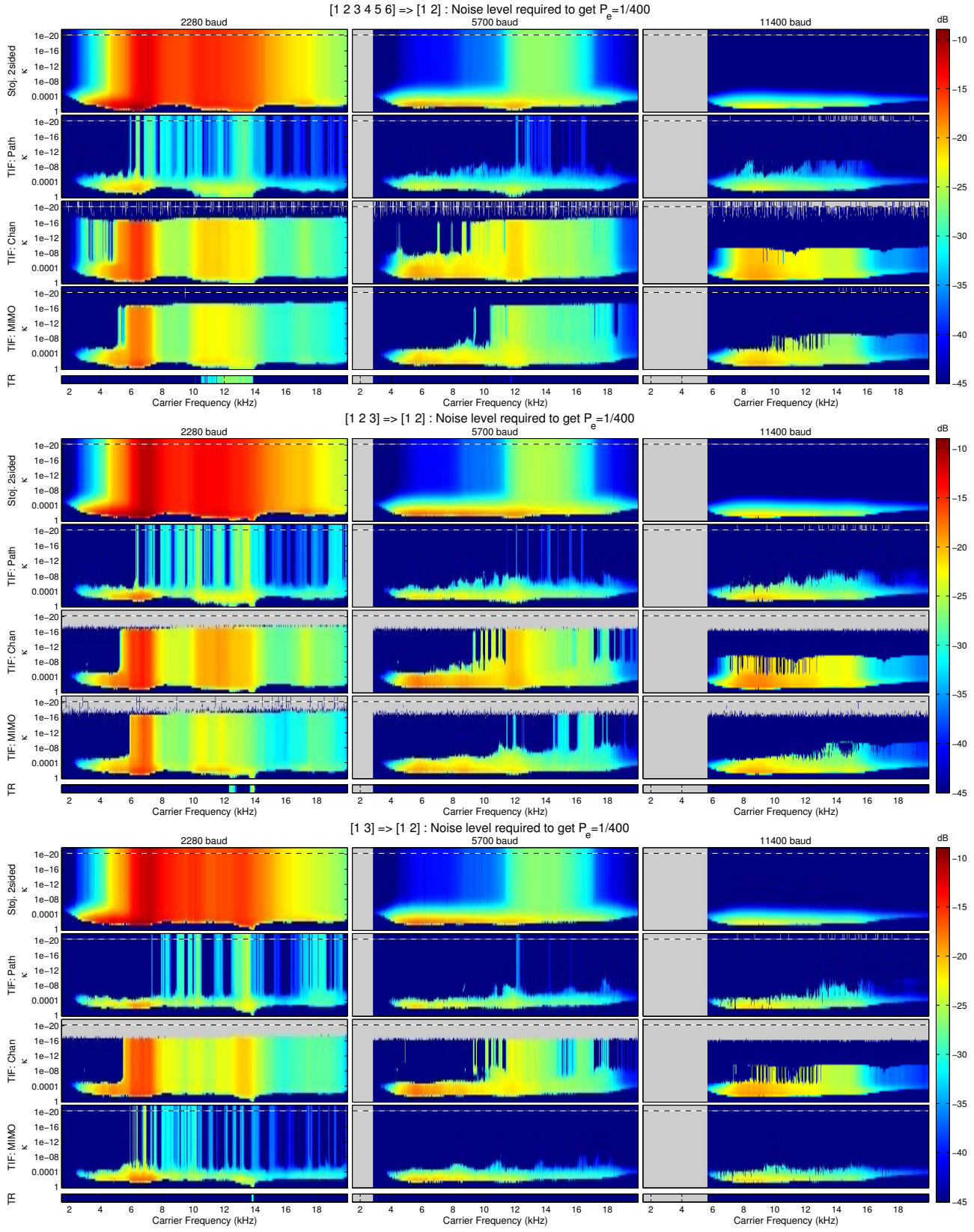


Figure 6.12: Channel noise required to achieve a standard deviation that results in an error rate of 1 in 400 for the parameter ranges presented in Table 6.1. Results for $\kappa = 0$ are presented in the plots in the top row of pixels above the dashed line.

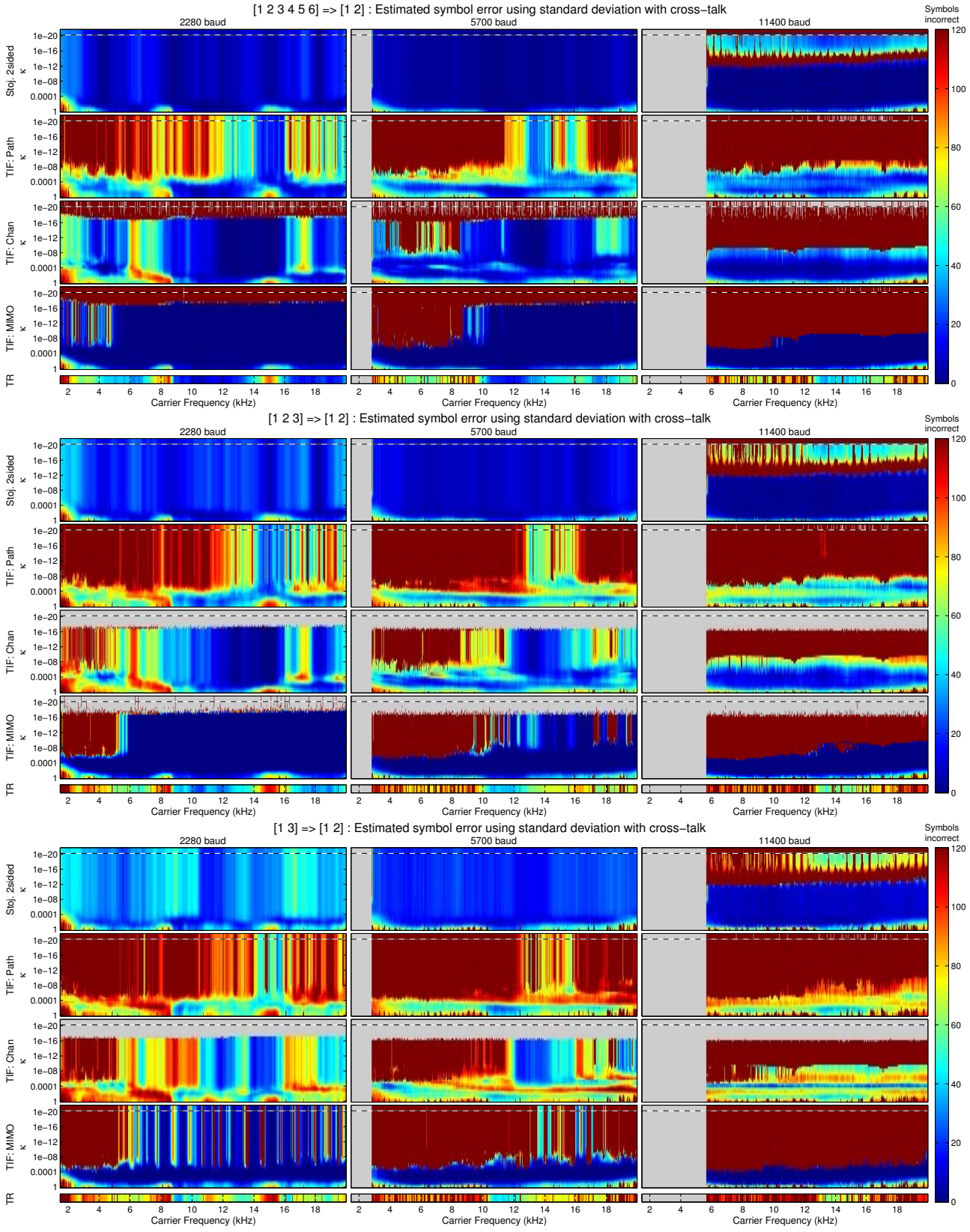


Figure 6.13: Estimate of symbol error derived from the standard deviation with the addition of cross-talk for the parameter ranges presented in Table 6.1. Results for $\kappa = 0$ are presented in the plots in the top row of pixels above the dashed line.

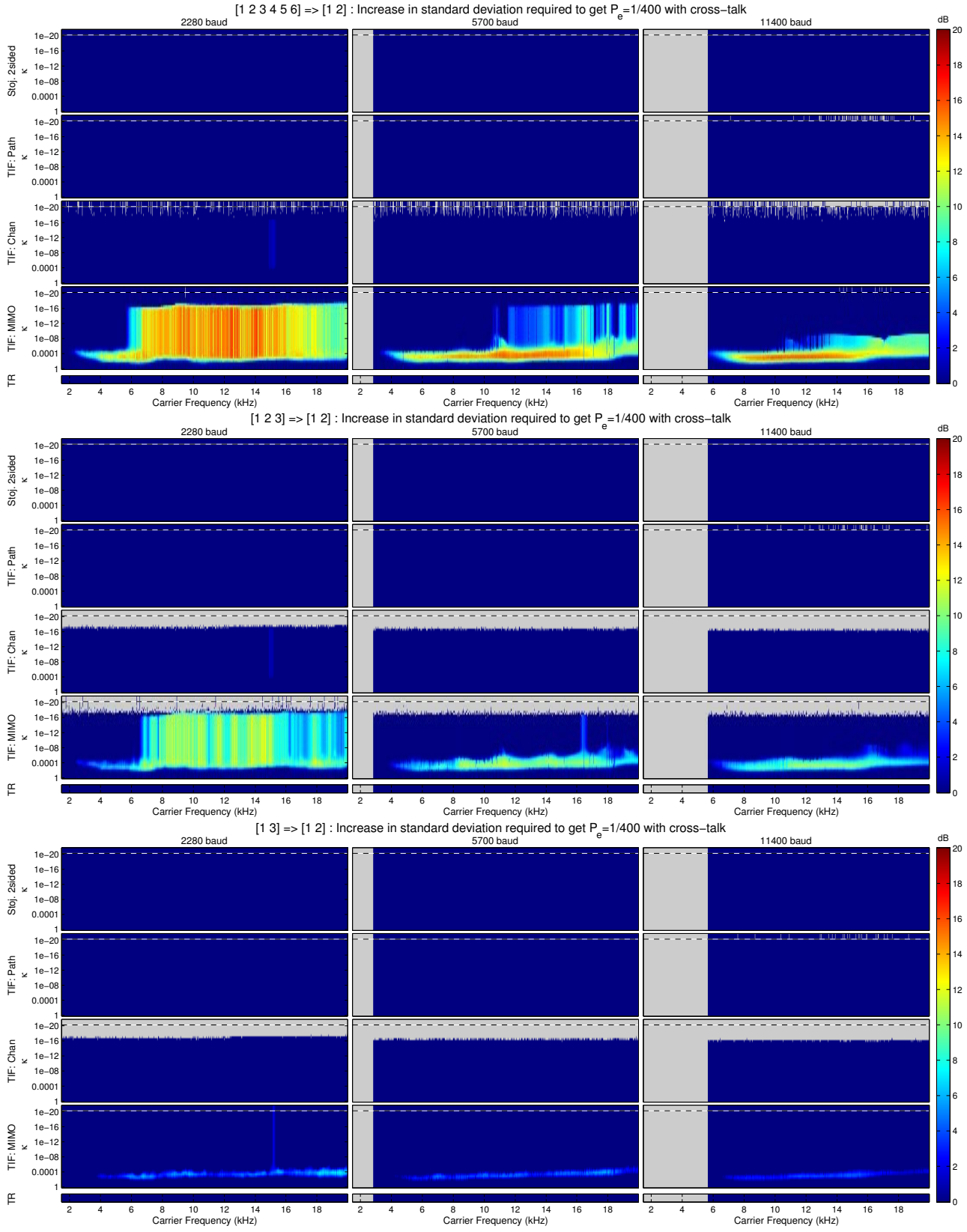


Figure 6.14: Increase in standard deviation required to achieve an error rate of 1 in 400 after the addition of cross-talk for the parameter ranges presented in Table 6.1. Results for $\kappa = 0$ are presented in the plots in the top row of pixels above the dashed line.

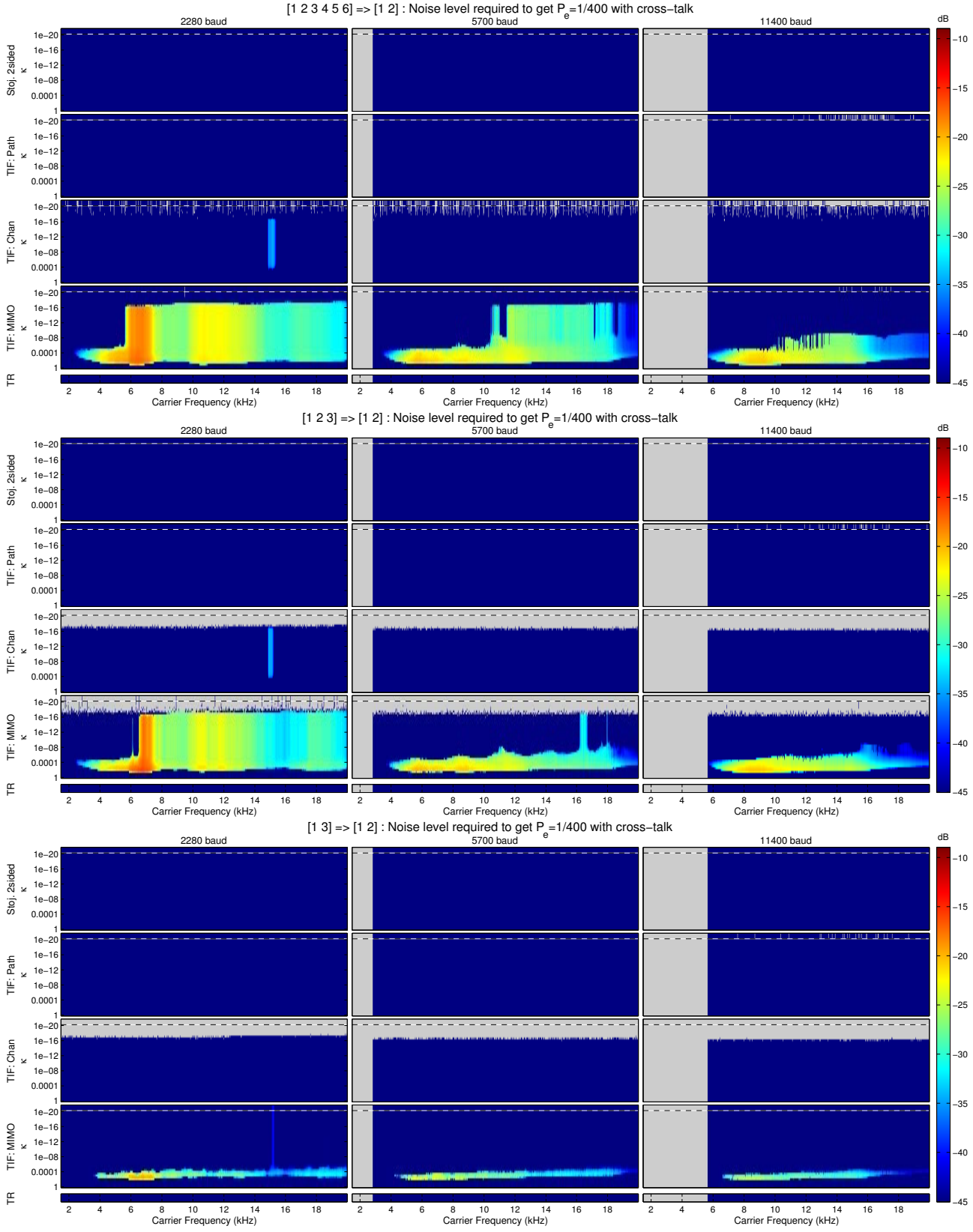


Figure 6.15: Channel noise required to achieve a standard deviation that results in an error rate of 1 in 400 after the addition of cross-talk for the parameter ranges presented in Table 6.1. Results for $\kappa = 0$ are presented in the plots in the top row of pixels above the dashed line.

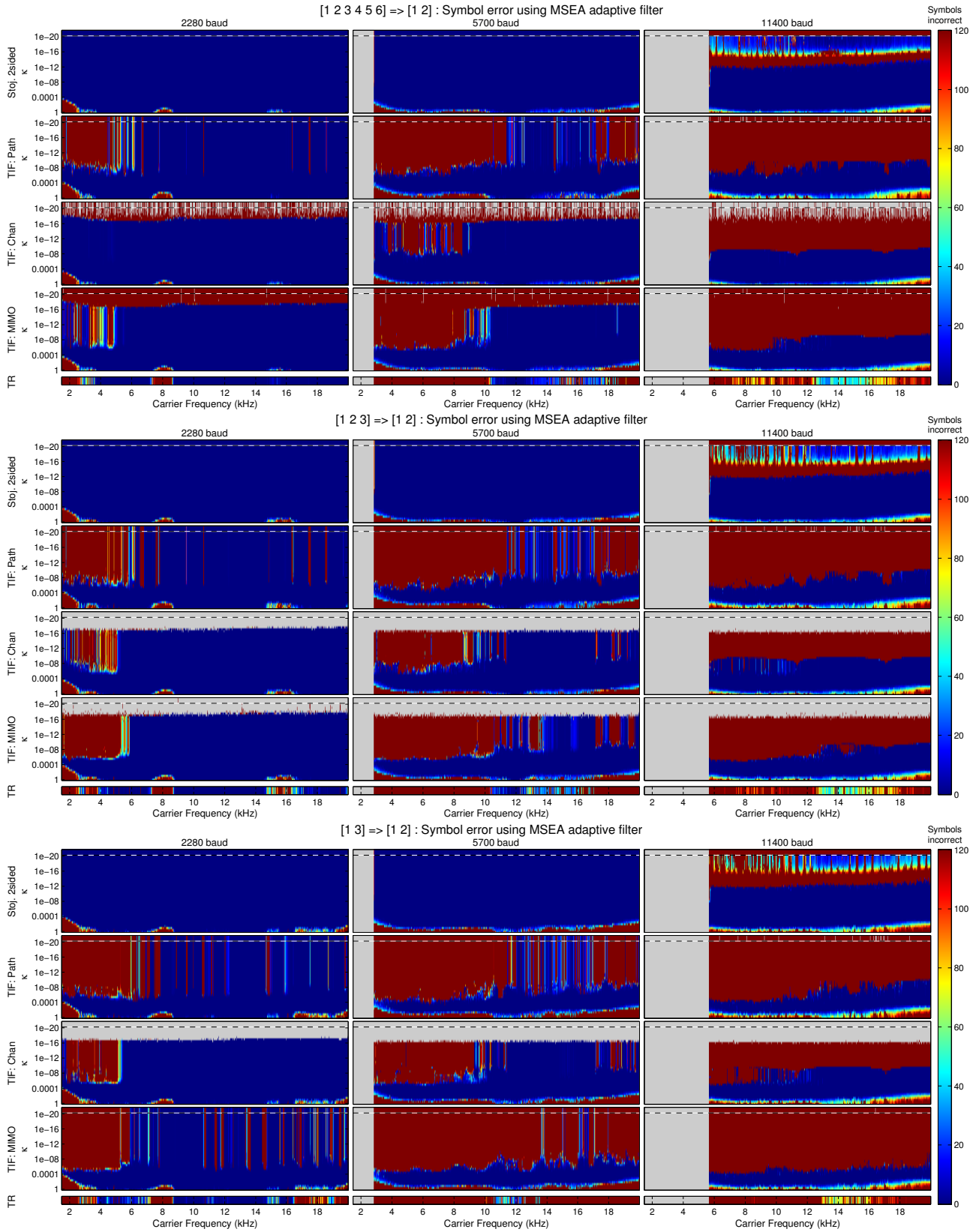


Figure 6.16: Symbol Error using LMS adaptive equaliser and no training sequence for the parameter ranges presented in Table 6.1. Results for $\kappa = 0$ are presented in the plots in the top row of pixels above the dashed line.

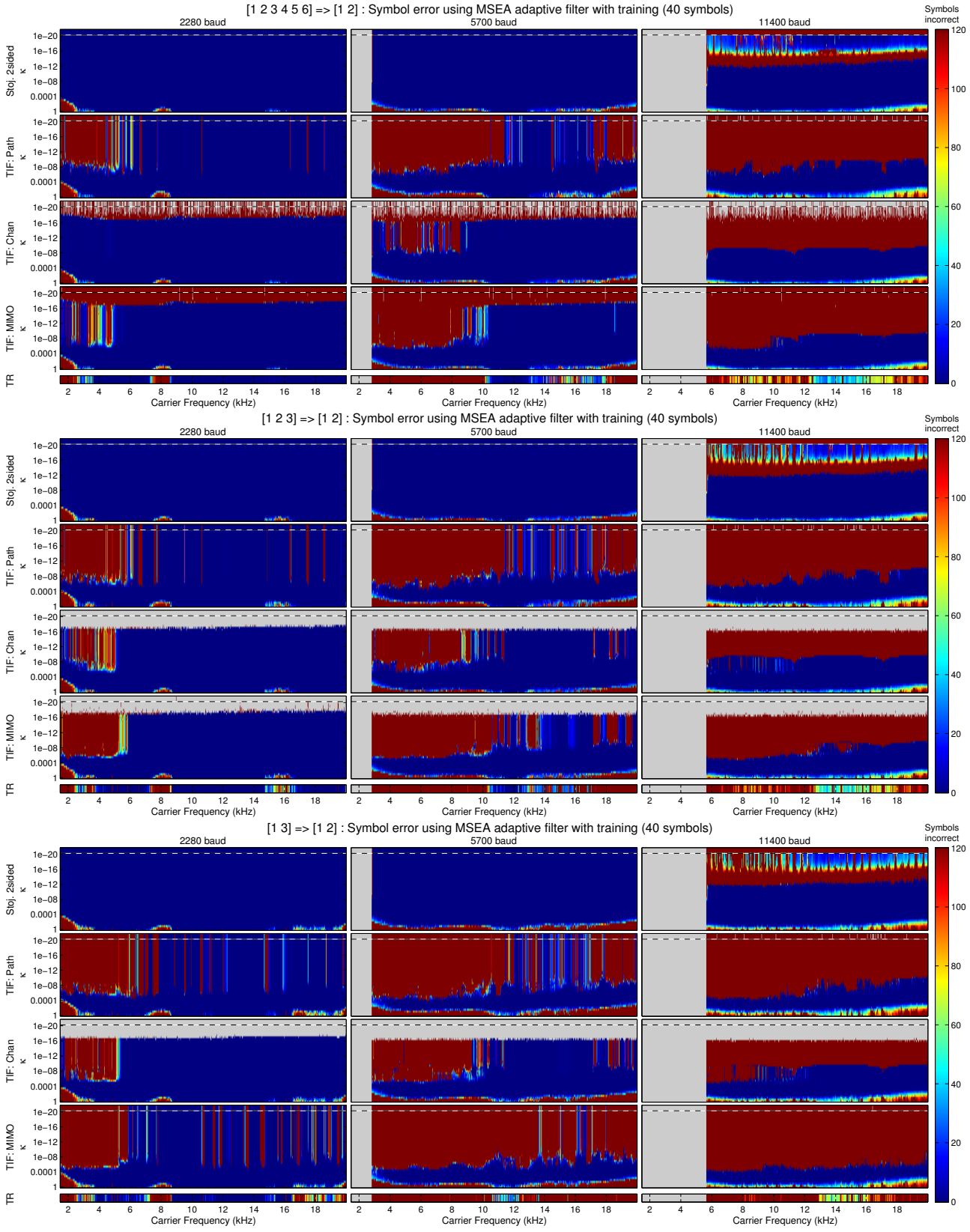


Figure 6.17: Symbol error using LMS adaptive equaliser and a training sequence of 40 symbols for the parameter ranges presented in Table 6.1. Results for $\kappa = 0$ are presented in the plots in the top row of pixels above the dashed line.

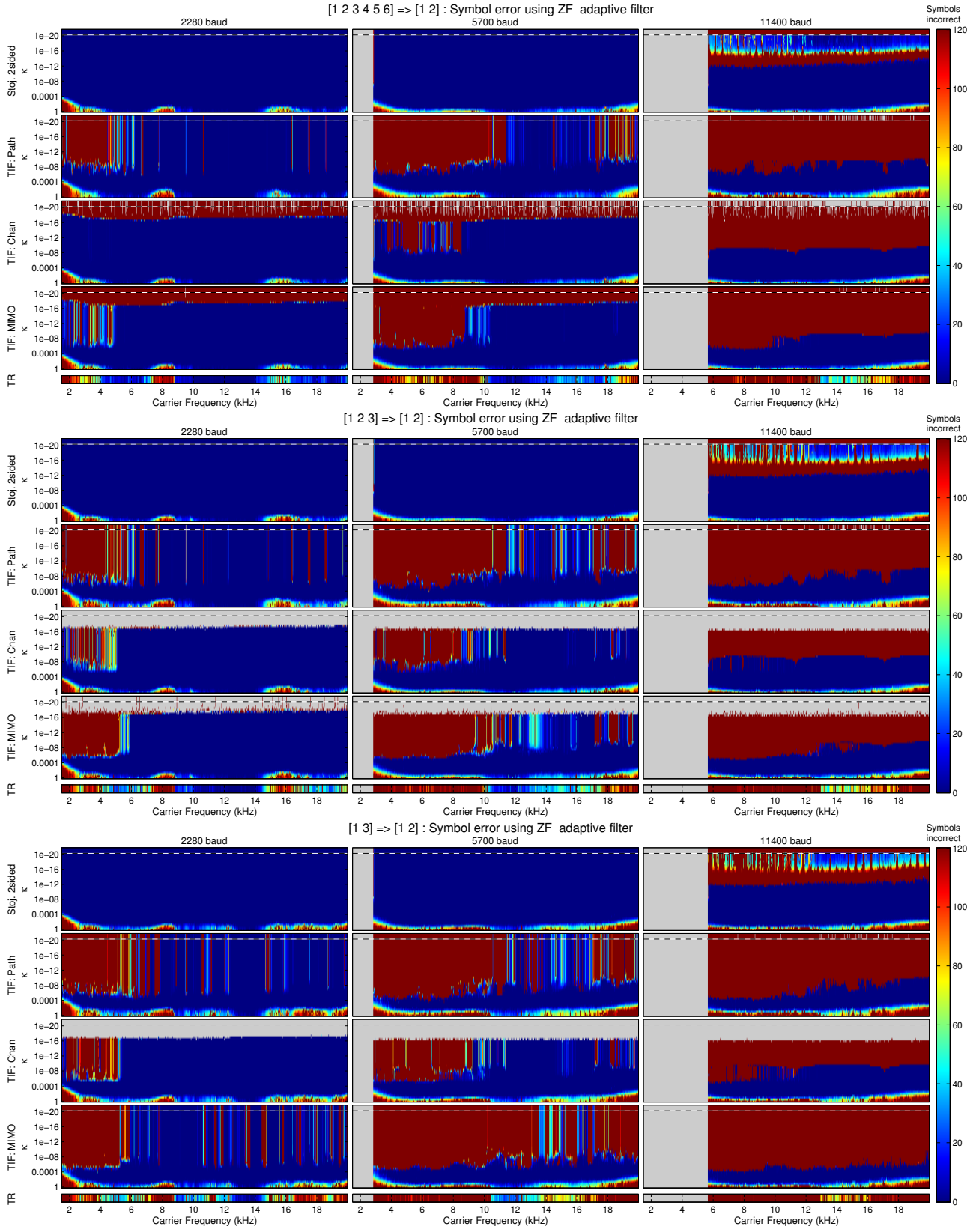


Figure 6.18: Symbol error using the zero-forcing adaptive equaliser and no training sequence for the parameter ranges presented in Table 6.1. Results for $\kappa = 0$ are presented in the plots in the top row of pixels above the dashed line.

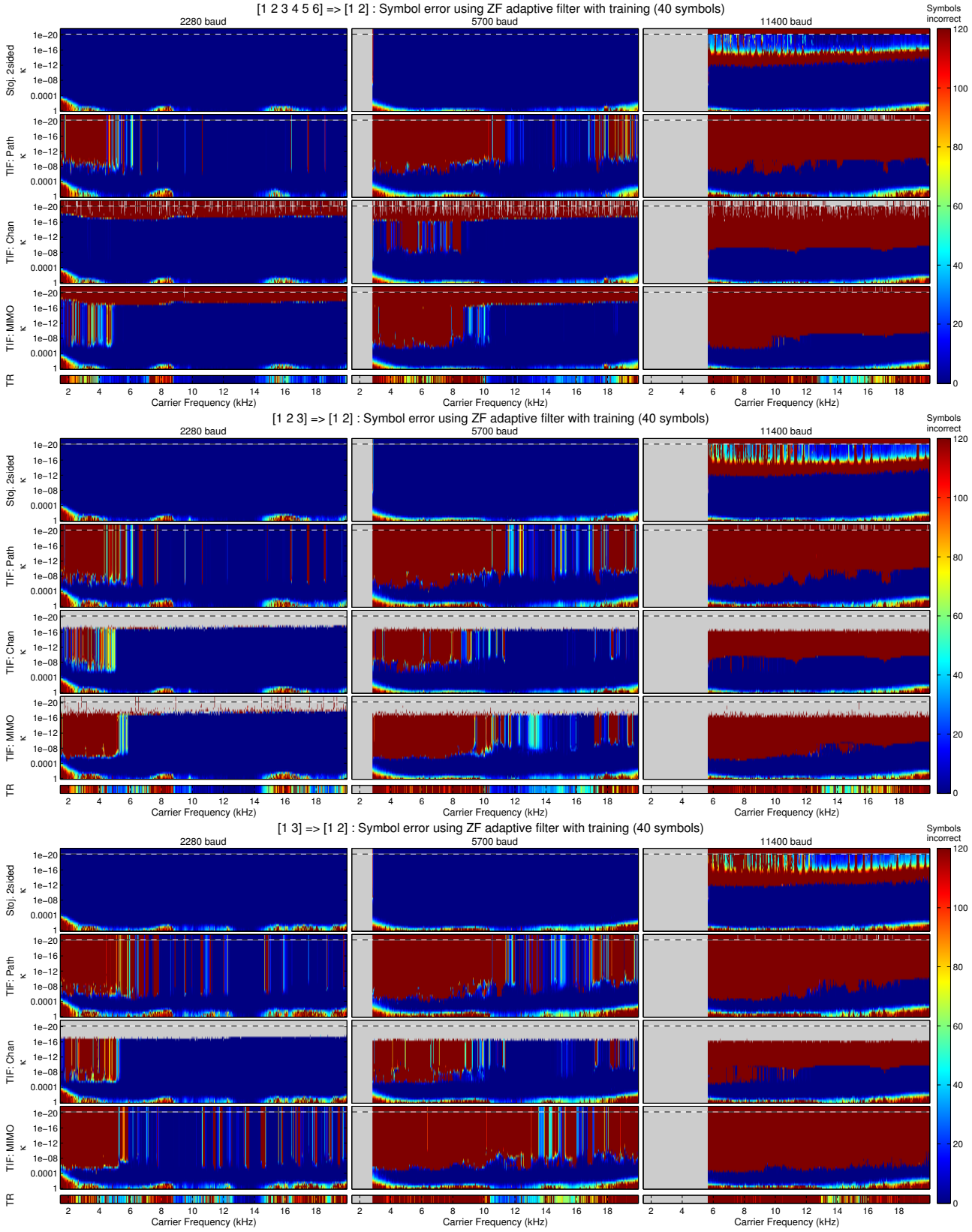


Figure 6.19: Symbol error using the zero-forcing adaptive equaliser and a training sequence of 40 symbols for the parameter ranges presented in Table 6.1. Results for $\kappa = 0$ are presented in the plots in the top row of pixels above the dashed line.

7 Conclusions and Future Work

7.1 Conclusions

The general aim of the research presented in this thesis is to examine the application of Tikhonov regularised inverse filtering to acoustic communication systems. The motivation for this research is to be able to apply the developments of this work to underwater acoustic communication systems.

In the literature two groups of researchers have examined the implementation of digital underwater acoustic communication systems for shallow water environments. One group investigated underwater acoustic communication from the basis of general digital communication theory [Baggeroer, 1984, Catipovic, 1990, Stojanovic, 1996, Kilfoyle and Baggeroer, 2000, Chitre et al., 2008]. The second group have looked at using an acoustic technique known as *time-reversal* and investigated means of integrating this technology with digital communication systems [Jackson and Dowling, 1991, Kuperman et al., 1998, Hodgkiss et al., 1999, Kim et al., 2001a]. Tikhonov regularised inverse filtering is observed to be similar to time-reversal in that both systems feature a focusing effect, and both aim to provide some form of reduced complexity in the design. The novel contributions of this thesis primarily relate to two particular developments: The influence of transducer sensitivity on Tikhonov inverse filtering, and an investigation of the performance of the Tikhonov inverse filter for use in communication systems. The details of these contributions and their implications are presented in the following two sections.

7.1.1 Influence of amplifier gain on Tikhonov inverse filter performance

During the implementation of Tikhonov inverse filtering in an experiment, it was found that the choice of amplifier sensitivities for the transmitting and receiving elements had an impact on the Tikhonov inverse filter performance. A technique was proposed to assist in compensating for the amplifier sensitivities. The influence of the amplifier sensitivities on Tikhonov regularised inverse filters was examined by mathematically considering the impact of

the sensitivities on the resulting filter co-efficients. Setting the sensitivities to their maximum for multi-channel systems does not always maximise the coherence between the input and output of the entire system.

The algorithm was tested on a set of channel responses that were used by Kirkeby et al. [1998] to demonstrate the functionality of the Tikhonov regularised inverse filter. The channel responses were scaled to emulate a system with poor sensitivities. Applying the adjustments to the sensitivities derived using the developed algorithm, the frequency response was found to be flatter, and the level of cross-talk reduced; indicating the algorithm was functioning as desired.

It was also shown that the impact the sensitivities has on the performance of the Tikhonov inverse filter varies depending on the regularisation parameter. By examining the Tikhonov inverse filter as the regularisation parameter approached zero and infinity, it was possible to gain insight into the performance of the inverse filters designs. It was found that if the regularisation was close to zero, the influence of the amplifier sensitivities on the total response was minimal, however as the regularisation parameter is increased, the influence becomes considerable. During the examination of various regularisation parameters it was also found that the time-reversal filter is equivalent to a Tikhonov regularised filter with infinite regularisation.

Since the time-reversal filter is equivalent to a Tikhonov regularised filter with infinite regularisation, many of the properties of time-reversal also apply to Tikhonov regularisation provided the regularisation is sufficiently large. In addition, if an optimisation strategy is created to find the optimal regularisation value for the Tikhonov regularised inverse filter, then the performance will always be greater or equal to that of the time-reversal filter since the time-reversal filter is included in the search space of the optimisation.

The ability to perform the compensation for the poor sensitivities in the digital domain was also investigated, however doing so resulted in a reduction of the magnitude of the channel response.

7.1.2 Implementation of Tikhonov inverse filtering for a communication system

An experimental investigation was performed to asses different methods of integrating various inverse filter designs into communication systems. An experiment was conducted that utilised Tikhonov regularised inverse filters in conjunction with an acoustic digital communication system in air. The experimental environment was designed to provide insight into how the filters might perform in a shallow underwater acoustic environment. A shallow underwater acoustic environment consists of a layer of water bounded by two surfaces: the sea surface, and the sea floor. These two surfaces result in the transmission undergoing many reflections when traversing through the

channel. In order to create a similar experimental environment, the channel used in the experiments was bounded by a number of surfaces, and multiple items were placed between the arrays to increase the number of reflections. Two arrays separated by 1 to 3 metres were placed in the environment and Tikhonov inverse filters used to create sets of signals that when transmitted would focus different information at each receiver.

In this work it was shown that the Tikhonov inverse filter and related filter design structures could be implemented according to three different classifications. The Tikhonov inverse filter was implemented according to each of these classifications and compared with each other and with two other filter designs: time-reversal filtering, and the Stojanovic two-sided filter.

The experiment demonstrated that the use of Tikhonov regularised inverse filter proved to have benefits over the other time-reversal and the Stojanovic two-sided filter. However, given the large number of parameters that could be altered it was difficult to gain a comprehensive comparison between the filters because of the time taken to run each experiment. The investigation was thus migrated to a simulation so that the experimental arrangement could be examined using various parameter configurations.

The parameters varied in the simulation were the selection of transmitter elements used, symbol rate, carrier frequency, regularisation parameter and filter type. The model of the channel response deviated from the physical communication experiment in that extraneous noise was not included in the model.

The simulation examined the response for a transmission that sent the information signal to a target receiver, and a null to the non-target receiver. A subsequent response was examined for a transmission that sent the information signal to the non-target receiver, and a null to the target receiver to examine the cross-talk at the target receiver. Extraneous noise was not included in the model because the results from a model containing no noise could be manipulated to provide an estimate of the results if noise had been included. Using these simulation results, the expected performance of the filter design for multi-channel communications could be derived.

The results obtained from the simulation demonstrated that the Stojanovic two-sided filter performed the best for single channel communications. However if the regularisation parameter was chosen correctly, the difference between the performance of the system using the Stojanovic two-sided filter and that using Tikhonov regularisation was small. The range of regularisation that provides effective results was found to decrease when the number of transmitters was reduced or the data rate increased. The Stojanovic two-sided filter was modified to include a regularisation parameter and it was found that at higher data rates, the filter performance improved when using regularisation.

The filter providing the best performance for multi-channel communica-

tions was the Tikhonov regularised inverse filter designed according to the *inverse by full MIMO* classification. This filter was the only one that was found to be functional for the system under investigation, providing zero symbol error over all the selections of transmitter elements, symbol rates and carrier frequencies. Some of the other filters could possibly be used in a multi-channel environment, however adaptive equalisers and other signal processing would be required to correct the symbol errors.

This research has satisfied its aim to investigate the influence of the Tikhonov inverse filter parameters on the communication system design, and its relative performance. The influences of the following were investigated: transducer placement, sensitivity of the transducers, parameters of the inverse filters, design structure of the Tikhonov regularised inverse filter, data rate, and carrier frequency. In addition, a technique has been developed to provide a means of compensating for poorly performing transducers from poor placement or incorrect sensitivities. Several implementations of the Tikhonov inverse filter were proposed and compared with two related filter designs: time-reversal and the Stojanovic two-sided filter. Given appropriate parameters, the Tikhonov regularised inverse filter successfully demonstrated good or comparable performance to the best filter tested for communication systems. In particular a Tikhonov regularised design out-performed the other filter designs tested with respect to multi-channel communication capability.

7.2 Recommendations for future work

The following sections describe potential lines of investigation to further the work presented in this thesis.

7.2.1 Methods for adapting the regularisation parameter

The simulations discussed in Chapter 6 provided good insight into the range of regularisation parameters for which the Tikhonov regularised inverse filter functions well. The addition of Tikhonov regularisation to the various inverse filtering systems influenced the magnitude and quality of the received signal and also the level of cross-talk to adjacent receivers. Observing the results from the simulations (Figures 6.5 to 6.19) the optimal choice of regularisation parameter appears to consistently be between 10^{-1} and 10^{-8}. However observing the level required to obtain an error of 1 in 400 with and without cross-talk (Figures 6.11 and 6.14) it is apparent that the optimal value can vary slightly depending on the channel response and the level of noise in the system.

It can be seen by examining the power of the received signal (Figure 6.6), that the magnitude of the received signal can be increased by increasing the regularisation parameter. For each scenario investigated, there was an optimal value for the regularisation parameter located between 10^{-1} and 10^{-8} that reduced the cross-talk power (Figure 6.8) and improved the quality of the received signal (Figures 6.9 and 6.11). If the regularisation strayed too far from this optimal area the cross-talk power would increase and the quality of the received symbols degrade. Future investigations into the inverse filter design could be made to develop a cost function that adapts the regularisation parameter to the conditions in which the filter design is to operate.

7.2.2 Adaptive channel estimates update using the symbol errors

The duration for which time-reversal provided sufficient filtering in a communication system was examined by Rouseff et al. [2001] for a shallow water environment for a number of environmental conditions. In calm conditions the communication continued to operate for some time, however in windy conditions or when the source was drifting, the symbols became less distinguishable over a much shorter duration. Flynn et al. [2004] investigated a means of using the detected symbols to update the co-efficients of the time-reversal filter to avoid completely halting the communication to obtain a fresh set of channel measurements. A similar technique could be applied to the Tikhonov regularised inverse filtering technique.

In order to test the implementation of channel updates, the waveguide experiment described in Chapter 5 could be modified to create a controlled dynamic environment that could provide repeatable variance of the channel environment through the use of motors to alter the position of the arrays or objects within the channel. Alternatively, a simulation with time-varying parameters could be used to assess the performance of the Tikhonov regularised inverse filtering technique.

7.2.3 Using the DORT technique to focus on each receiver

In Section 3.2.1, the technique of using a decomposition of the temporal operator (DORT) was described for matched field processing systems which was able to create signals that could focus on any of the desired reflective targets in an environment. An avenue for future work could involve using the same technique on floating receiver locations with a central base station. Each of these floating target receiver locations might retransmit an initial

probe signal, and the DORT technique could then be used to create signals specifically targeted for each receiver location.

7.2.4 Variable range focusing

Song et al. [1998] demonstrated that the focal zone of time-reversal could be moved by shifting the frequency response of the channel measurement used in the time-reversal process. The phenomenon could be combined with a communication system to predict the response at different locations for travelling vessels. The phenomenon could also be used in conjunction with Tikhonov regularised inverse filtering to zero the signal at different ranges to tighten the spatial focus of the signal.

7.2.5 Automatic channel MIMO reduction

When deploying MIMO communication systems for use in the underwater environment, some environments might provide a favourable channel in which the MIMO system can effectively transmit various signals to each receiver with little cross-talk or signal distortion, whilst in other environments there could be more cross-talk interference or signal degradation for the same arrangement. It would be desirable to design a communication system that can detect the quality of the environment and dynamically adapt the number of channels over which to transmit in order to maximise the channel capacity.

7.2.6 Using the adjoint operator to eliminate cross-talk

In the development of the Tikhonov regularised inverse filter design, Kirkeby et al. [1996b] showed that given the z-transform representation of the filter

$$\mathbf{H}(z) \;=\; \left[\mathbf{C}^{\mathrm{T}}(z^{-1})\mathbf{C}(z) + \kappa\mathbf{I}\right]^{-1}\mathbf{C}^{\mathrm{T}}(z), \tag{7.1}$$

the filter can also be expressed as

$$\mathbf{H}(z) \;=\; \frac{\mathrm{adj}\left[\mathbf{C}^{\mathrm{T}}(z^{-1})\mathbf{C}(z) + \kappa\mathbf{I}\right]\mathbf{C}^{\mathrm{T}}(z^{-1})}{\det\left[\mathbf{C}^{\mathrm{T}}(z^{-1})\mathbf{C}(z) + \kappa\mathbf{I}\right]} \tag{7.2}$$

where the adj[ ] and det[ ] operators are the adjoint and determinant operators respectively. A property of both the adjoint and determinant operators is that the elements of the output matrix consist of multiplications of the elements of the input matrix. Thus if the elements of the matrix consist of finite length filters, then the resulting output will also consist of finite length filters. Thus any instability that results from the Tikhonov regularised filter design is the result of the inversion of the determinant. Since the determinant results in a scalar value, the portion of the filter design that

achieves cross-talk cancellation must be in the numerator. Thus it would appear that a finite-length cross-talk canceller can theoretically be formed using the numerator of the filter,

$$\mathbf{H}_{a_1}(z) \;=\; \text{adj}\left[\mathbf{C}^{\mathrm{T}}(z^{-1})\mathbf{C}(z^{-1}) + \kappa\mathbf{I}\right]\mathbf{C}^{\mathrm{T}}(z^{-1}) \tag{7.3}$$

and setting κ to zero. Whilst this might result in a stable cross-talk cancelling filter, the length of the total system response may be very long. The length of the response of the Tikhonov regularised inverse filter can be seen to be controlled by the application of the additional filter

$$H_{a_2}(z) \;=\; \frac{1}{\det\left[\mathbf{C}^{\mathrm{T}}(z^{-1})\mathbf{C}(z) + \kappa\mathbf{I}\right]}, \tag{7.4}$$

being the denominator of the Tikhonov regularised inverse filter. It would be of interest to observe the functionality of a filter design that replaces the inverse operator with a time-reversal equivalent, resulting in the computationally less expensive filter

$$H_{a_2}(z) \;=\; (\det\left[\mathbf{C}^{\mathrm{T}}(z^{-1})\mathbf{C}(z) + \kappa\mathbf{I}\right])^{*}. \tag{7.5}$$

References

J.B. Allen, D.A. Berkley, and J. Blaert. Multimicrophone signal-processing techniques to remove room reverberation from speech signals. *Journal of the Acoustical Society of America*, 62(4):912–915, October 1977. See page 54.

A.B. Baggeroer. Acoustic telemetry - an overview. *IEEE Journal of Oceanic Engineering*, 9(4):229–235, October 1984. See pages 1, 33, 34, and 157.

H.P. Bucker. Sound propagation in a channel with lossy boundaries. *Journal of the Acoustical Society of America*, 48(5):1187–1194, 1970. See page 11.

H.P. Bucker. Use of calculated sound fields and matched-field detection to locate sound sources in shallow water. *Journal of the Acoustical Society of America*, 59(2):368–373, February 1976. See page 42.

J.V. Candy, A.W. Meyer, A.J. Poggio, and B.L. Guidry. Time-reversal processing for an acoustic communications experiment in a highly reverberant environment. *Journal of the Acoustical Society of America*, 115(4):1621–1631, April 2004. See pages 68 and 69.

J.V. Candy, A.J. Poggio, D.H. Chambers, B.L. Guidry, C.L. Robbins, and C.A. Kent. Multi-channel time-reversal processing for acoustic communications in a highly reverberant environment. *Journal of the Acoustical Society of America*, 118(4):2339–2354, October 2005. See pages 68 and 69.

D. Carsten, J.-C. Aime, and M. Fink. One-channel time-reversal in chaotic cavities: Experimental results. *Journal of the Acoustical Society of America*, 105(2):618–625, February 1999. See page 44.

D. Cassereau and M. Fink. Time-reversal of ultrasonic fields-part iii: Theory of the closed time-reversal cavity. *IEEE Transactions on Ultrasonics, FerroElectrics, and Frequency Control*, 39(5):579–592, September 1992. See page 40.

D. Cassereau and M. Fink. Focusing with plane time-reversal mirrors: An efficient alternative to closed cavities. *Journal of the Acoustical Society of America*, 94(4):2373–2386, October 1993. See page 40.

J.A. Catipovic. Performace limitations in underwater acoustic telemetry. *IEEE Journal of Oceanic Engineering*, 15(3):205–216, July 1990. See pages 1, 33, 34, 35, and 157.

B.S. Cazzolato, P. Nelson, P. Joseph, and R.J. Brind. Numerical simulation of optimal deconvolution in a shallow-water environment. *Journal of the Acoustical Society of America*, 110(1):170–185, July 2001. See pages 2, 61, 70, 71, 72, 123, 126, and 137.

D. H. Chambers and A.K. Gautesen. Time reversal for a single spherical scatterer. *Journal of the Acoustical Society of America*, 109(6):2616–2624, June 2001. See page 48.

M. Chitre, S. Shahabudeen, and M. Stojanovic. Underwater acoustic communications and networking: Recent advances and future challenges. *The Spring*, 42(1):103–116, 2008. See pages 1, 33, 35, and 157.

C.S. Clay. Waveguides, arrays, and filters. *GeoPhysics*, 31(3):501–505, June 1966. See pages 37 and 50.

C.S. Clay. Optimum time domain signal transmission and source location in a waveguide. *Journal of the Acoustical Society of America*, 81(3):660–664, March 1987. See pages 42 and 49.

C.S. Clay and Li Saimu. Time domain signal transmission and source location in a waveguide: Matched filter and deconvolution experiments. *Journal of the Acoustical Society of America*, 83(4):1377–1383, April 1988. See page 60.

P. Damaske. Head-related two-channel stereophony with loudspeaker reproduction. *Journal of the Acoustical Society of America*, 50(4):1109–1115, 1971. See page 54.

A. Derode, P. Roux, and M. Fink. Robust acoustic time reversal with high-order multiple scattering. *Physics Review Letters*, 75(23):4206–4210, December 1995. doi: 10.1103/PhysRevLett.75.4206. See page 42.

A. Derode, A. Tourin, and M. Fink. Ultrasonic pulse compuression with one-bit time reversal through multiple scattering. *Journal of Applied Physics*, 85(9):6343–6352, May 1999. See pages 44 and 71.

C. Dorme and M. Fink. Focusing in transmit–receiver mode through inhomogeneous media: The time reversal matched filter approach. *Journal of the Acoustical Society of America*, 98(2):1155–1162, August 1995. See page 60.

C. Dorme and M. Fink. Ultrasonic beam steering through inhomogeneous layers with a time reversal mirror. *IEEE Transactions on Ultrasonics, FerroElectrics, and Frequency Control*, 43(1):167–175, January 1996. See pages 40 and 41.

D.R. Dowling. Acoustic pulse compression by passive phase-conjugation. *Journal of the Acoustical Society of America*, 95(3):1450–1458, March 1994. See pages 49 and 63.

D.R. Dowling and D.R. Jackson. Narrow-band performance of phase-conjugate arrays in dynamic random media. *Journal of the Acoustical Society of America*, 91(6):3257–3277, June 1992. See page 42.

C. Draeger, D. Cassereau, and M. Fink. Theory of time-reversal process in solids. *Journal of the Acoustical Society of America*, 102(3):1289–1295, September 1997. See page 54.

dSPACE. *DS1104 R&D Controller Board - RTLib Reference*. dSPACE GmbH, Technologiepark 25, 33100 Paderborn, Germany, release 4.1 edition, March 2004. See pages 179 and 186.

P.M. Dumuid and B.S. Cazzolato. Experimental results of time reversal and optimal inverse filtering performed in a one dimensional waveguide. In *145th Meeting: Acoustical Society of America*, volume 114, pages 2407–2408. Acoustical Society of America, October 2003. See page 109.

P.M. Dumuid, B.S. Cazzolato, and A.C. Zander. A comparison of filter design structures for multi-channel acoustic communication systems. *Journal of the Acoustical Society of America*, 123(1):174–185, January 2008. See page 121.

R.M. Dunbar. The performace of magnetic loop transmitter-receiver systems submerged in the sea. *The Radio and Electronic Engineer*, 42(10):457–463, October 1972. See page 1.

G.F. Edelmann, T. Akal, W.S. Hodgkiss, S. Kim, W.A. Kuperman, and H.C. Song. An initial demonstration of underwater acoustic communication using time reversal. *IEEE Journal of Oceanic Engineering*, 27(3):602–609, July 2002. See page 67.

G.F. Edelmann, H.C. Song, S. Kim, W.S. Hodgkiss, W.A. Kuperman, and T. Akal. Underwater acoustic communications using time reversal. *IEEE Journal of Oceanic Engineering*, 30(4):852–864, October 2005. See pages 67 and 68.

S.J. Elliott. *Signal Processing for Active Control*. Academic Press, 1st ed. edition, 2001. See page 96.

P.C. Etter. *Underwater acoustic modeling - Principles, Techniques and Applications*. E & FN Spon, 2nd edition, 1996. See pages xiii, 1, 5, 7, 8, 9, 11, 12, and 14.

A.C. Fannjiang. Time reversal communications in rayleigh-fading broadcast channels with pinholes. *Physics Letters A*, 353(5):389–397, 2006. See page 69.

M. Fink. Time reversal of ultrasonic fields - part i: Basic principles. *IEEE Transactions on Ultrasonics, FerroElectrics, and Frequency Control*, 39(5): 555–566, September 1992. See pages xiv, 37, 39, 45, and 49.

M. Fink, G. Montaldo, and M. Tanter. Time-reversal acoustics in biomedical engineering. *Annual Review of Biomedical Engineering*, 5:465–497, August 2003. See page 53.

J.L. Flanagan and R.C. Lummis. Signal processing to reduce multipath distortion in small rooms. *Journal of the Acoustical Society of America*, 47(6):1475–1481, 1970. See page 54.

J.A. Flynn, J.A. Ritcey, D. Rouseff, and W.L.J. Fox. Multichannel equalization by decision-directed passive phase conjugation: Experimental results. *IEEE Journal of Oceanic Engineering*, 29(3):824–836, July 2004. See pages 65 and 161.

B. Gardner and K. Martin. HRTF measurements of a KEMAR dummy-head microphone. Technical Report 280, MIT Media Lab Perceptual Computing, MIT Media Lab, E15-401D, Cambridge, MA 02139, May 1994. See page 82.

G.H. Golub, M. Heath, and G. Wahba. General cross-validation as a method for choosing a good ridge parameter. *Technometrics*, 21(2):215–223, May 1979. See page 59.

T. Goursolle, S. Dos Santos, O. Bou Matar, and S. Calle. Non-linear based time reversal acoustic applied to crack detection: Simulation and experiments. *International Journal of Non-Linear Mechanics*, 43(3):170–177, 2008. See page 54.

P.C. Hansen. *Rank-deficient and discrete ill-posed problems: numerical aspects of linear inversion*. SIAM Monographs on Mathematical Modeling and Computation. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998. ISBN 0-89871-403-6. See pages 58 and 59.

S. Haykin. *Communication Systems*. John Wiley & Sons, Inc., fourth edition, 2001. See page 26.

M. Heinemann, A. Larraza, and K.B. Smith. Experimental studies of applications of time-reversal acoustics to noncoherent underwater communications. *Journal of the Acoustical Society of America*, 113(6):3111–3116, June 2003. See page 67.

W.J. Higley, P. Roux, and W.A. Kuperman. Relationship between time reversal and linear equalization in digital communications (l). *Journal of the Acoustical Society of America*, 120(1):35–37, July 2006. See pages 59 and 126.

W.S. Hodgkiss, H.C. Song, W.A. Kuperman, T. Akal, C. Ferla, and D. R. Jackson. A long-range and variable focus phase-conjugation experiment in shallow water. *Journal of the Acoustical Society of America*, 105(3): 1597–1604, March 1999. See pages 2, 41, 51, 63, and 157.

O. Ikeda. An image reconstruction algorithm using phase conjugation for diffraction-limited imaging in an inhomogeneous medium. *Journal of the Acoustical Society of America*, 85(4):1602–1606, April 1989. See pages 37 and 45.

D.R. Jackson and D.R. Dowling. Phase conjugation in underwater acoustics. *Journal of the Acoustical Society of America*, 89(1):171–181, January 1991. See pages xiv, 2, 37, 38, 39, 40, 50, 63, 76, and 157.

F.B. Jensen, W.A. Kuperman, M.B. Porter, and H. Schmidt. *Computational Ocean Acoustics*. American Institute of Physics, New York, 2000. See pages xiii, 5, 11, 12, and 13.

E. Kerbrat, D. Clorennec, C. Prada, D. Royer, D. Cassereau, and M. Fink. Detection of cracks in a thin air-filled hollow cylinder by application of the dort method to elastic components of the echo. *Ultrasonics*, 40:715–720, 2002. See page 54.

E. Kerbrat, C. Prada, D. Cassereau, and M. Fink. Imaging in the presence of grain noise using the decomposition of the time reversal operator. *Journal of the Acoustical Society of America*, 113(3):1230–1240, March 2003. See page 49.

D.B. Kilfoyle and A.B. Baggeroer. The state of the art in underwater acoustic telemetry. *IEEE Journal of Oceanic Engineering*, 25(1):4–27, January 2000. See pages 1, 33, 35, and 157.

J.S. Kim and K.C. Shin. Multiple focusing with adaptive time-reversal mirror. *Journal of the Acoustical Society of America*, 115(2):600–606, February 2004. See page 71.

J.S. Kim, H.C. Song, and W.A. Kuperman. Adaptive time-reversal mirror. *Journal of the Acoustical Society of America*, 109(5):1817–1825, May 2001a. See pages 2, 41, 49, 63, 71, and 157.

S. Kim, G.F. Edelman, W.A. Kuperman, W.S. Hodgkiss, H.C. Song, and T. Akal. Spatial resolution of time-reversal arrays in shallow water. *Journal of the Acoustical Society of America*, 110(2):820–829, August 2001b. See pages 42 and 52.

Y. Kim and P. A. Nelson. Spatial resolution limits for the reconstruction of acoustic source strength by inverse methods. *Journal of Sound and Vibration*, 265(3):583–608, August 2003. See page 59.

Y. Kim and P.A. Nelson. Estimation of acoustic source strength within a cylindrical duct by inverse methods. *Journal of Sound and Vibration*, 275(1–2):391–413, 2004a. ISSN 0022-460X. doi: DOI:10.1016/j.jsv.2003.06.032. URL `http://www.sciencedirect.com/science/article/B6WM3-4B3MS2F-1/2/eecb44ea196ed02ac15447e87203201e`. See page 60.

Y. Kim and P.A. Nelson. Optimal regularisation for acoustic source reconstruction by inverse methods. *Journal of Sound and Vibration*, 275(3–5):463–487, August 2004b. See page 59.

Y. Kim, O. Deille, and P.A. Nelson. Crosstalk cancellation in virtual acoustic imaging systems for multiple listeners. *Journal of Sound and Vibration*, 297(1–2):251–266, October 2006. See page 60.

O. Kirkeby and P.A. Nelson. Reproduction of plane wave sound fields. *Journal of the Acoustical Society of America*, 94(5):2992–3000, November 1993. See page 59.

O. Kirkeby, P.A. Nelson, H. Hamada, and F. Orduña Bustamante. Fast deconvolution of multi-channel systems using regularisation. Technical Report 255, Institute of Sound and Vibration Research, Southhampton S017 1BJ, England, April 1996a. See pages 2 and 56.

O. Kirkeby, P.A. Nelson, F. Orduna-Bustamante, and H. Hamada. Local sound field reproduction using digital signal processing. *Journal of the Acoustical Society of America*, 100(3):1584–1593, September 1996b. See pages 59 and 162.

O. Kirkeby, P.A. Nelson, H Hamada, and F. Orduña Bustamante. Fast deconvolution of multichannel systems using regularization. *IEEE Transactions on Speech and Audio Processing*, 6(2):189–195, March 1998. See pages xiv, xv, 57, 58, 82, 83, 91, and 158.

W.A. Kuperman, W.S. Hodgkiss, H.C. Song, T. Akal, C. Ferla, and D.R. Jackson. Phase conjugation in the ocean: Experimental demonstration of an acoustic time-reversal mirror. *Journal of the Acoustical Society of America*, 103(1):25–40, January 1998. See pages xiv, 2, 13, 44, 49, 50, 51, 52, 63, 75, and 157.

T. Leutenegger and J. Dual. Detection of defects in cylindrical structures using a time reversal method and a finite-difference approach. *Ultrasonics*, 40:721–725, 2002. See page 54.

T. Leutenegger and J. Dual. Non-destructive testing of tubes using a time reversal simulation (trns) method. *Ultrasonics*, 41:811–822, 2004. See page 54.

S. Li and C.S. Clay. Optimum time domain signal transmission and source location in a waveguide: Experiments in an ideal wedge waveguide. *Journal of the Acoustical Society of America*, 82(4):1409–1417, October 1987. See page 49.

J.F. Lingevitch, H.C. Song, and W.A. Kuperman. Time reversed reverberation focusing in a waveguide. *Journal of the Acoustical Society of America*, 111(6):2609–2614, June 2002. See page 49.

M. Miyoshi and Y. Kaneda. Inverse filtering in room acoustics. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 36(2):145–152, Feburary 1988. See page 56.

G. Montaldo, M. Tanter, and M. Fink. Real time inverse filter focusing through iterative time reversal. *Journal of the Acoustical Society of America*, 115(2):768–775, February 2004. See pages 58 and 126.

N. Mordant, C. Prada, and M. Fink. Highly resolved detection and selective focusing in a waveguide using the D.O.R.T. method. *Journal of the Acoustical Society of America*, 105(5):2634–2642, May 1999. See page 48.

S.T. Neely and J.B. Allen. Invertibility of a room impulse response. *Journal of the Acoustical Society of America*, 66(1):165–169, July 1979. See page 56.

P.A. Nelson. Active control of acoustic fields and the reproduction of sound. *Journal of Sound and Vibration*, 177(4):447–477, November 1994. See page 59.

P.A. Nelson and S.H. Yoon. Estimation of acoustic source strength by inverse methods: Part I, conditioning of the inverse problem. *Journal of Sound and Vibration*, 233(4):643–668, 2000. See page 59.

P.A. Nelson, H. Hamada, and J. Elliott. Adaptive inverse filters for stereophonic sound reproduction. *IEEE Transactions on Signal Processing*, 40 (7):1621–1632, 1992. See page 56.

P.A. Nelson, F. Orduña Bustamante, and H. Hamada. Inverse filter design and equalization zones in multichannel sound reproduction. *IEEE Transactions on Speech and Audio Processing*, 3(3):185–192, May 1995. See pages 56 and 59.

H.W. Park, H. Sohn, K.H. Law, and C.R. Farrar. Time reversal active sensing for health monitoring of composite plate. *Journal of Sound and Vibration*, 302:50–66, 2007. See page 54.

A. Parvulescu. Matched-signal ("MESS") processing by the ocean. *Journal of the Acoustical Society of America*, 98(2):943–960, August 1995. See pages 2, 36, and 50.

A. Parvulescu and C.S. Clay. Reproducibility of signal transmission in the ocean. *The Radio and Electronic Engineer*, 29:223–228, 1965. See page 63.

M. Pernot, J.-F. Aubry, M. Tanter, A.L. Boch, and M. Fink. Ultrasonic transcranial brain therapy: first in vivo clinical investigation on 22 sheep using adaptive focusing. *IEEE Ultrasonics Symposium*, 2(23):1013–1016, 2004. See page 53.

R. P. Porter and A. J. Devaney. Generalized holography and the inverse source problems. *Journal of the Optical Society of America*, 72:327–330, 1982. See page 40.

C. Prada and M. Fink. Eigenmodes of the time reversal operator: a solution to selective focusing in multiple-target media. *Wave Motion*, 20(2):151–163, 1994. See page 45.

C. Prada and M. Fink. Separation of interfering acoustic scattered signals using the invariants of the time-reversal operator. application to Lamb waves characterization. *Journal of the Acoustical Society of America*, 104 (2):801–807, August 1998. See page 48.

C. Prada, F. Wu, and M. Fink. The iterative time reversal mirror: A solution to self-focusing in the pulse echo mode. *Journal of the Acoustical Society of America*, 90(2):1119–1129, August 1991. See pages xiv, 37, 38, 39, 45, and 46.

C. Prada, J.-L. Thomas, and M. Fink. The iterative time reversal process: Analysis of the convergence. *Journal of the Acoustical Society of America*, 97(1):62–71, January 1995. See pages xiv, 45, 47, and 48.

C. Prada, S. Manneville, D. Spoliansky, and M. Fink. Decomposition of the time reversal operator: Detection and selective focusing on two scatterers. *Journal of the Acoustical Society of America*, 99(4):2067–2076, April 1996. See pages 48 and 76.

J.G. Proakis. *Digital Communications*. McGraw-Hill Series in Electrical and Computer Engineering. McGraw-Hill, fourth edition, 2001. See pages xiv, 16, 17, 18, 20, 26, 28, 29, 30, 50, 128, and 131.

J.H Rose, M. Bilgen, P. Roux, and M. Fink. Time-reversal mirrors and rough surfaces: Theory. *Journal of the Acoustical Society of America*, 106 (2):716–723, August 1999. See page 53.

D. Rouseff. Intersymbolic interference in underwater acoustic communications using time-reversal signal processing. *Journal of the Acoustical Society of America*, 117(2):780–788, Febuary 2005. See page 66.

D. Rouseff, D.R. Jackson, W.L.J. Fox, C.D. Jones, J.A. Ritcey, and D.R. Dowling. Underwater acoustic communication by passive-phase conjugation: Theory and experimental results. *IEEE Journal of Oceanic Engineering*, 26(4):821–831, October 2001. See pages 65 and 161.

P. Roux and M. Fink. Time reversal in a waveguide: Study of the temporal and spatial focusing. *Journal of the Acoustical Society of America*, 107(5): 2418–2429, may 2000. See pages 44, 52, 95, and 96.

P. Roux, B. Roman, and M. Fink. Time-reversal in an ultrasonic waveguide. *Applied Physics Letters*, 70(14):1811–1813, April 1997. doi: 10.1063/1.118730. URL `http://link.aip.org/link/?APL/70/1811/1`. See pages xiv, 44, 45, 50, and 95.

P. Roux, A. Derode, A. Peyre, A. Tourin, and M. Fink. Acoustical imaging through a multiple scattering medium using a time-reversal mirror. *Journal of the Acoustical Society of America - Acoustics Research Letters Online*, 107(2):L7–L12, Feburary 2000. See page 54.

P. Roux, W.A. Kuperman, W.S. Hodgkiss, H.C. Song, and T. Akal. A nonreciprocal implementation of time reversal in the ocean. *Journal of the Acoustical Society of America*, 116(2):1009–1015, August 2004. See pages 68 and 69.

D. Ruiz. A scaling algorithm to equilibrate both rows and columns norms in matrices. Technical Report RAL-TR-2001-034, Rutherford Appleton Laboratory, September 2001. See page 80.

K. G. Sabra and D. R. Dowling. Blind deconvolution in ocean waveguides using artificial time reversal. *Journal of the Acoustical Society of America*, 116(1):262–271, July 2004. See page 53.

K. G. Sabra, S. R. Khosla, and D. R. Dowling. Broadband time-reversing array retrofocusing in noisy environments. *Journal of the Acoustical Society of America*, 111(2):823–830, Feburary 2002. See pages 50 and 53.

O. Shimbo and M. Celebiler. The probability of error due to intersymbol interference and Gaussian noise in digital communication systems. *IEEE*

Transactions on Communication Technology, 19(2):113–119, April 1971. See page 131.

A Silva, S. Jesus, J. Gomes, and V. Barrosso. Underwater acoustic communication using a "virtual" electronic time-reversal mirror approach. Lyon, France, 2000. Fifth European Conference on Underwater Acoustics, ECUA 2000. See page 65.

B. Sklar. *Digital Communications - Fundamentals and Application*. Prentice Hall PTR, 2001. See pages xiii, 15, 16, and 17.

A. Skretting and C.C. Leroy. Attenuation between 200 Hz and 10 kHz. *Journal of the Acoustical Society of America*, 49(1):276–282, 1971. See page 5.

K.B. Smith, A.M. Abrantes, and A. Larraza. Examination of time-reversal acoustics in shallow water and application to noncoherent underwater communications. *Journal of the Acoustical Society of America*, 113(6):3095–3110, June 2003. See page 67.

H.C. Song and S.M. Kim. Retrofocusing techniques in a waveguide for acoustic communications (letter). *Journal of the Acoustical Society of America*, 121(6):3277–3279, June 2007. See pages 69 and 70.

H.C. Song, W.A. Kuperman, and W.S. Hodgkiss. A time-reversal mirror with variable range focusing. *Journal of the Acoustical Society of America*, 103 (6):3234–3240, June 1998. See pages xiv, 41, 42, 43, and 162.

H.C. Song, W.A. Kuperman, W.S. Hodgkiss, T. Akal, and C. Ferla. Iterative time reversal in the ocean. *Journal of the Acoustical Society of America*, 105(6):3176–3184, June 1999. See page 52.

H.C. Song, W.S. Hodgkiss, W.A. Kuperman, M. Stevenson, and T. Akal. Improvement of time reversal communications using adaptive channel equalizers. *IEEE Journal of Oceanic Engineering*, 31(2):487–496, 2006a. See page 69.

H.C. Song, W.S. Kodgkiss, W.A. Kuperman, W.J. Higley, K. Raghukuar, and T. Akal. Spatial diversity in passive time reversal communications. *Journal of the Acoustical Society of America*, 120(4):2067–2076, October 2006b. See page 66.

H.C. Song, P. Roux, W.S. Hodgkiss, W.A. Kuperman, T. Akal, and M. Stevenson. Multiple input / multiple output coherent time reversal communications in a shallow water acoustic channel. *IEEE Journal of Oceanic Engineering*, 31(1):170–178, January 2006c. See page 69.

H.C. Song, W.S. Hodgkiss, W.A. Kuperman, T. Akal, and M. Stevenson. Multiuser communications using passive time reversal. *IEEE Journal of Oceanic Engineering*, 32(4):915–926, October 2007. See page 66.

H.C. Song, W.A. Kuperman, and W.S. Hodgkiss. Basin-scale time reversal communications. *Journal of the Acoustical Society of America*, 125(1): 212–217, January 2009. See page 66.

M. Stojanovic. Recent advances in high-speed underwater acoustic communications. *IEEE Journal of Oceanic Engineering*, 21(2):125–136, April 1996. See pages 1, 33, 35, and 157.

M. Stojanovic. Retrofocusing techniques for high rate acoustic communications. *Journal of the Acoustical Society of America*, 117(3):1173–1185, march 2005. See pages iii, xvi, 4, 69, 70, 72, 107, 111, 123, 124, 125, 126, 128, 129, 130, 136, 137, 138, 139, 140, 141, 159, and 160.

M. Stojanovic, J.A. Catipovic, and J.G. Proakis. Adaptive multichannel combining and equalization for underwater acoustic communications. *Journal of the Acoustical Society of America*, 94(3):1621–1631, September 1993. See page 35.

M. Stojanovic, J.A. Catipovic, and J.G. Proakis. Phase-coherent digital communications for underwater acoustic channels. *IEEE Journal of Oceanic Engineering*, 19(1):100–111, January 1994. See page 35.

M. Stojanovic, J.A. Catipovic, and J.G. Proakis. Reduced-complexity spatial and temporal processing of underwater acoustic communication signals. *Journal of the Acoustical Society of America*, 98(2):961–972, August 1995. See page 35.

M. Tanter, J.-L. Thomas, and M. Fink. Time reversal and the inverse filter. *Journal of the Acoustical Society of America*, 108(1):223–234, July 2000. See pages 61 and 77.

M. Tanter, J.-F. Aubry, J. Gerber, J.-L. Thomas, and M. Fink. Optimal focusing by spatio-temporal inverse filter. I. Basic principles. *Journal of the Acoustical Society of America*, 110(1):37–47, July 2001. See page 77.

T. Tanter, J.-L. Thomas, and M. Fink. Focusing and steering through absorbing and aberrating layers: Application to ultrasonic propagation through the skull. *Journal of the Acoustical Society of America*, 103(5):2403–2410, May 1998. See pages 41, 53, and 60.

J.-L. Thomas and M. Fink. Ultrasonic beam focusing through tissue inhomogeneities with a time reversal mirror: application to transskull therapy.

IEEE Transactions on Ultrasonics, FerroElectrics, and Frequency Control, 43(6):1122–1129, 1996. See pages xiv, 60, and 61.

J.-L. Thomas, F. Wu, and M. Fink. Time reversal focusing applied to lithotripsy. *Ultrasonic Imaging*, 18(2):106–121, 1996. See page 53.

W.H. Thorpe. Analytic description of the low-frequency attenuation coefficient. *Journal of the Acoustical Society of America*, 42(1):270, 1967. See page 6.

I. Tolstoy and C.S. Clay. *Ocean Acoustics - Theory and Experiment in Underwater Sound*. American Institute of Physics, 2nd edition, 1987. See pages xiii, 5, 7, 9, and 10.

A. Van der Sluis. Condition numbers and equilibration of matrices. *Numerische Mathematik*, 14(1):14–23, 1969. See page 79.

F. Vignon, J.-F. Aubry, A. Saez, M. Tanter, D Cassereau, G. Montaldo, and M. Fink. The stokes relations linking time reversal and the inverse filter. *Journal of the Acoustical Society of America*, 119(3):1335–1346, March 2006. See page 62.

T.C. Yang. Temporal resolutions of time-reversal and passive-phase conjugation for underwater acoustic communications. *IEEE Journal of Oceanic Engineering*, 28(2):229–245, April 2003. See page 65.

T.C. Yang. Differences between passive-phase conjugation and decision-feedback equalizer for underwater acoustic communications. *IEEE Journal of Oceanic Engineering*, 29(2):472–487, April 2004. See page 65.

T.C. Yang. Correlation-based decision-feedback equalizer for underwater acoustic communications. *IEEE Journal of Oceanic Engineering*, 30(4): 865–880, October 2005. See page 66.

S. Yon, M. Tanter, and M. Fink. Sound focusing in rooms. II. the spatio-temporal inverse filter. *Journal of the Acoustical Society of America*, 114 (6):3044–3052, December 2003a. See page 62.

S. Yon, M. Tanter, and M. Fink. Sound focusing in rooms: The time-reversal approach. *Journal of the Acoustical Society of America*, 113(3):1533–1543, March 2003b. See page 62.

K. Yoo and T.C. Yang. Broadband source localization in shallow water in the presense of internal waves. *Journal of the Acoustical Society of America*, 106(6):3255–3269, December 1999. See page 53.

Appendices

A Program developed for the experiment and simulation

A.1 The dSpace development system

In this section the code used to perform the experiments is presented[1]. The experiment was conducted using a dSPACE ds1104 board. The dSPACE ds1104 board is a PCI card that contains its own CPU, 8 A/D converters and 8 D/A converters. The board was developed by dSPACE for the rapid prototyping of Digital Signal Processing (DSP) systems that are designed using Simulink, a program that is provided with MATLAB. dSPACE provide software that converts DSP models designed in Simulink to a program that is downloaded and executed on the card. Monitoring routines are implemented on the card to allow the host computer to read values from the program as it executes.

When investigating the capabilities of the dSPACE card for use in this research, it was desired that programs would execute at a speed that would allow the A/D converters and D/A converters to operate at a speed sufficient for the acoustic signals. However, early investigations showed that the use of Simulink to develop programs for the card was unable to run at the speeds desired. (It should be noted that since these initial investigations, recent upgrades to the dSPACE system may provide faster speeds.) dSPACE also provided a library to enable users to write programs in C, along with tools to compiled user written C code an downloaded the program to the card. A library of functions was also available to simplify the programming of the device, and allow MATLAB to control the device via reading and writing to / from memory addresses, or sending "host interrupt" events that execute portions of the code [dSPACE, 2004].

This chapter presents the C-code and MATLAB scripts used to interact and run with the experiments.

[1]Please note that the code shown in this thesis has been slightly modified (such as variable renaming, and removal of redundant code) from the original source used in the experiments to ease the readability and has not since been tested. If you wish to obtain the original code, please contact the author (pmdumuid@gmail.com).

A.2 Code used for the experiment using inverse filter designs in an air-acoustic channel

In this section, the code used for the experiment used to investigate the ability to achieve digital transmission in conjunction with inverse filter design over an air-acoustic channel. The code consists of a C program, `DuctExperimentDSPACEProgram.c`, that is compiled and downloaded to run on the dSPACE DS1104 controller board and two main MATLAB script, `DuctExperimentCreateTransmissionSignal.m`, and `DuctExperimentPlayAndPostprocess.m`. The MATLAB script, `DuctExperimentCreateTransmissionSignal.m`, is used to create the transmission signals, and the `DuctExperimentPlayAndPostprocess.m` script is used play the signal, and perform post-process on the received signal. A number of other *helper scripts*, were developed that contain functions required by these main scripts. Some of these functions are specific to this experiment, whilst others have been re-used for the third experiment. The scripts specific to this experiment are presented in Section A.2.4, whilst the functions that have been re-used are included in the thesis MATLAB library given in Section A.4.

A.2.1 DuctExperimentDSPACEProgram.c

A.2.1.1 Program listing

```
1  /* DuctExperimentDSPACEProgram.c ************************************************************************
2
3  Author: Pierre Dumuid
4  Description:
5
6     This program is developed to be used for the duct experiment conducted as part of the thesis done
7     by Pierre Dumuid. The program is used to play signals on the outputs, and record the response.
8
9  ********************************************************************************************************/
10
11 #include <Brtenv.h>
12 #include "math.h"
13
14 /* GLOBAL VARIABLES:START */
15
16 /* Data block for play and record MATLAB communication. */
17 #define block_size 1024*100
18 Float64 data_block[32*block_size];
19
20 /* Variable used to let the host computer set the digital I/O bit.  */
21 UInt16 io_bits_write  = 0;
22 /* Variable used to let the host computer read the digital I/O bit. */
23 UInt32 io_bits_status = 0;
24
25 /* Program status and control. */
26 UInt16 program_status    = 0; // Used to indicate status of program
27 UInt16 program_control   = 0; // Used to control what to do in the interrupt program.
28 UInt16 main_loop_control = 0; // Used to control what to do in the loop in the main() function.
```

```
29
30 /* Used for the level trigger functionality. */
31 Float64 trigger_level = 0.1;
32
33 /* LED flash routine variables. */
34 UInt32 led_flash_counter = 0;
35 UInt32 led_flash_bits = 0xE00;
36
37 /* Recording / playback settings. */
38 Float64 sample_period = 0.0001;
39
40 UInt32 play_sample_length    = 400;
41 UInt32 record_sample_length = 400;
42
43 /* Array to indicate which channels to play and record on.  The first element in the array defines
44    the number of channels to use, and the subsequent values, the actual channels to use. */
45 UInt16 play_channels[9]   = {6, 1, 2, 3, 4, 5, 6 ,7, 8};
46 UInt16 record_channels[9] = {6, 1, 2, 3, 4, 5, 6, 7, 8};
47
48 /* Gains to apply when playing and recording. */
49 Float64 play_channel_gains[8]   = {1, 1, 1, 1, 1, 1, 1 ,1};
50 Float64 record_channel_gains[8] = {1, 1, 1, 1, 1, 1, 1 ,1};
51
52 /* GLOBAL VARIABLES:END */
53
54
55 void Play_Stepped_SIMO() {
56     /*
57       Plays a single signal on each channel one-by-one, and records the response from the recording
58       channels. The data_block is used as follows:
59
60       0                       -  1*play_sample_length-1 : Sample to play
61       1*play_sample_length  -  2*play_sample_length-1 : Channel 1 response for playing on channel 1
62       2*play_sample_length  -  3*play_sample_length-1 : Channel 2 response for playing on channel 1
63       ...
64       6*play_sample_length  -  7*play_sample_length-1 : Channel 6 response for playing on channel 1
65       7*play_sample_length  -  8*play_sample_length-1 : Time samples for playing on channel 1
66       9*play_sample_length  - 10*play_sample_length-1 : Channel 1 response for playing on channel 2
67       10*play_sample_length - 11*play_sample_length-1 : Channel 2 response for playing on channel 2
68       ...
69       14*play_sample_length - 15*play_sample_length-1 : Channel 6 response for playing on channel 2
70       15*play_sample_length - 16*play_sample_length-1 : Time samples for playing on channel 2
71       ...
72       ...
73       etc.
74
75       NOTE: The above is an example for recording on 6 channels, if recording on a different number
76       of channels, the offsets for the data is equivalently adjusted.
77
78       i.e. record_channel_offset =
79        play_sample_length + (record_channel_index*(number_of_record_channels + 1)*play_sample_length
80     */
81
82     float   current_time, next_sample_time;
83     UInt16  adc_mux_scan_table[4] = {1, 2, 3, 4};
84     UInt16  play_channel_index   , play_channel;
85     UInt16  record_channel_index, record_channel;
86     UInt32  sample_index;
87
88     Float64 adc_mux_values[6];
89
90     program_status = 1;
91
92     for (play_channel_index = 0; play_channel_index < play_channels[0]; play_channel_index++) {
93         play_channel = play_channels[play_channel_index + 1];
```

```
94
95          ds1104_tic_start();
96
97          next_sample_time = ds1104_tic_read() + sample_period;
98
99          /* Loop through all the samples */
100         for (sample_index = 0; sample_index < play_sample_length; sample_index++) {
101
102             /* Wait for the time of the next sample. */
103             do {
104                 master_cmd_server();   current_time = ds1104_tic_read();
105             } while (current_time < next_sample_time);
106             next_sample_time += sample_period;
107
108             /* Write the sample to play to the corresponding channel. */
109             ds1104_dac_write(play_channel,
110                              play_channel_gains[play_channel_index] * data_block[sample_index]);
111
112             /* Read the ADC mux, and start the ADCs. */
113             ds1104_adc_read_mux(adc_mux_scan_table, 4, adc_mux_values);
114             ds1104_adc_start(DS1104_ADC2 | DS1104_ADC3 | DS1104_ADC4 | DS1104_ADC5);
115
116             /* Record the sample time in the data_block. Index is determined as follows:
117                1. Length of the play sample
118                2. Length of the data for the previously played channels.
119                3. Offset for the time data block
120                4. Offset for the current sample.
121             */
122
123             data_block[
124                         play_sample_length                                          // 1
125                         + play_sample_length * play_channel_index * (record_channels[0] + 1) // 2
126                         + play_sample_length * record_channels[0]                   // 3
127                         + sample_index                                              // 4
128                         ] = current_time;
129
130             /* Record the ADC values for the record channels in the data_block. Index is determined
131              * as follows:
132                1. Length of the play sample
133                2. Length of the data for the previously played channels.
134                3. Offset for the current record channel.
135                4. Offset for the current sample.
136             */
137             for(record_channel_index = 0;
138                 record_channel_index < record_channels[0];
139                 record_channel_index++) {
140
141                 record_channel = record_channels[record_channel_index + 1] - 1;
142
143                 if (record_channel < 4) {
144                     data_block[
145                                 play_sample_length                                  // 1
146                                 + play_sample_length*play_channel_index*(record_channels[0] + 1) // 2
147                                 + play_sample_length*record_channel_index           // 3
148                                 + sample_index                                      // 4
149                                 ] =
150                         adc_mux_values[record_channel]*record_channel_gains[record_channel];
151                 } else {
152                     data_block[
153                                 play_sample_length                                  // 1
154                                 + play_sample_length*play_channel_index*(record_channels[0] + 1) // 2
155                                 + play_sample_length*record_channel_index           // 3
156                                 + sample_index                                      // 4
157                                 ] =
158                         ds1104_adc_read_ch(record_channel + 1)*record_channel_gains[record_channel];
```

```
159                 }
160             }
161         }
162     }
163     program_status = 2;
164 }
165
166
167 void Play_MIMO() {
168     /*
169       Plays the single signal defined data_block on the playing channels and record the
170       response. The data_block is used as follows:
171
172       0                       - 1*play_sample_length-1    : Output 1 Sample
173       1*play_sample_length    - 2*play_sample_length-1    : Output 2 Sample
174         ...
175    (N-1)*play_sample_length   - N*play_sample_length-1    : Output N Sample
176
177    After record_offset:
178       0                       - 1*record_sample_length-1 : Input 1 Signal
179       1*record_sample_length - 2*record_sample_length-1 : Input 2 Signal
180         ...
181    (M-1)*record_sample_length - M*record_sample_length-1 : Input 3 Signal
182
183    After time_offset:
184       0                       - max_sample_length         : Sampling times
185
186        where
187          record_offset      = N*play_sample_length
188          time_offset        = N*play_sample_length + M*record_sample_length
189          max_sample_length = max(record_sample_length, play_sample_length)
190     */
191
192     UInt16  adc_mux_scan_table[4] = {1, 2, 3, 4};
193     Float64 adc_mux_values[6];
194
195     UInt16 play_channel_index, record_channel_index, record_channel;
196     UInt32 sample_index;
197
198     float current_time, next_sample_time, start_time;
199     UInt32 record_offset, time_offset;
200     UInt32 max_sample_length;
201
202     program_status = 1;
203
204     record_offset = play_channels[0] * play_sample_length;
205     time_offset   = record_offset + record_channels[0] * record_sample_length;
206
207     if (record_sample_length > play_sample_length)
208         max_sample_length = record_sample_length;
209     else
210         max_sample_length = play_sample_length;
211
212     ds1104_tic_start();
213
214     start_time = ds1104_tic_read();
215
216     /* Loop through all the samples */
217     for (sample_index = 0; sample_index < max_sample_length; sample_index++) {
218
219         /* Define the time for the next sample (wait 5 samples_period's before starting) */
220         next_sample_time = start_time + (sample_index + 5)*sample_period;
221
222         /* Wait for the time of the next sample */
223         do {
```

```
224             current_time = ds1104_tic_read();
225         } while (current_time < next_sample_time);
226 
227         /* Send the signals to the DAC converters. */
228         if (sample_index <= play_sample_length) {
229             for (play_channel_index = 0;
230                  play_channel_index < play_channels[0];
231                  play_channel_index++)
232                 ds1104_dac_write(play_channels[play_channel_index + 1],
233                                  data_block[play_channel_index*play_sample_length + sample_index]
234                                  * play_channel_gains[play_channel_index]);
235             ds1104_dac_strobe();
236         }
237 
238         /* Read the values on the ADC converters */
239         if (sample_index <= record_sample_length) {
240             /* Read the ADC mux, and start the ADCs. */
241             ds1104_adc_read_mux(adc_mux_scan_table, 4, adc_mux_values);
242             ds1104_adc_start(DS1104_ADC2 | DS1104_ADC3 | DS1104_ADC4  | DS1104_ADC5);
243 
244             /* Record the sample time in the data_block. */
245             data_block[time_offset + sample_index] = current_time;
246 
247             /* Record the values on the ADC converters */
248             for(record_channel_index = 0;
249                 record_channel_index < record_channels[0];
250                 record_channel_index++) {
251                 record_channel = record_channels[record_channel_index + 1] - 1;
252                 if (record_channel < 4) {
253                     data_block[record_offset
254                                + record_channel_index * record_sample_length
255                                + sample_index
256                                ] =
257                         adc_mux_values[record_channel]
258                         * record_channel_gains[record_channel];
259                 } else {
260                     data_block[record_offset
261                                + record_channel_index * record_sample_length
262                                + sample_index
263                                ] =
264                         ds1104_adc_read_ch(record_channel + 1)
265                         * record_channel_gains[record_channel];
266                 }
267             }
268         }
269     }
270     program_status = 2;
271 }
272 
273 
274 /* Define the IIOF2 service interrupt routine */
275 void Host_Interrupt_Service() {
276     float adc_mux_value1;
277     program_status = 1;
278     switch (program_control) {
279     case 1:
280         break;
281     case 2: // Wait for a trigger level to be reached
282         program_status = 1;
283         do {
284             ds1104_adc_delayed_start(DS1104_ADC_CH1);
285             ds1104_adc_mux(1);
286             adc_mux_value1 = ds1104_adc_read_ch(1);
287             master_cmd_server();
288         } while (adc_mux_value1 < trigger_level);
```

```
289         program_status = 2;
290         break;
291     case 3:
292         Play_Stepped_SIMO();
293         break;
294     case 4:
295         Play_MIMO();
296         break;
297     case 5:
298         break;
299     case 99:
300         main_loop_control = 0;
301         break;
302     default:
303         break;
304     }
305     program_status = 2;
306 }
307
308
309 /* Register the service interrupt routine */
310 void Initialise_Interrupt_Service() {
311     /* set the interrupt vector */
312     ds1104_set_interrupt_vector( DS1104_INT_HOST,
313                                  (DS1104_Int_Handler_Type) Host_Interrupt_Service,
314                                  SAVE_REGS_ON);
315
316     /* enable interrupt IIOF2 */
317     ds1104_enable_hardware_int(DS1104_INT_HOST);
318
319     /* global interrupt enable */
320     DS1104_GLOBAL_INTERRUPT_ENABLE();
321 }
322
323
324 void main(void) {
325     UInt16  adc_mux_scan_table[4] = {1, 2, 3, 4};
326     Float64 adc_mux_values[4];
327
328     ds1104_init();
329     Initialise_Interrupt_Service();
330     ds1104_dac_reset();
331     ds1104_dac_init(DS1104_DACMODE_TRANSPARENT);
332
333     ds1104_bit_io_init(DS1104_DIO0_OUT | DS1104_DIO1_OUT |
334                        DS1104_DIO2_OUT | DS1104_DIO3_OUT |
335                        DS1104_DIO4_OUT | DS1104_DIO5_OUT |
336                        DS1104_DIO6_OUT | DS1104_DIO7_OUT |
337                        DS1104_DIO8_OUT | DS1104_DIO9_OUT |
338                        DS1104_DIO10_OUT | DS1104_DIO11_OUT |
339                        DS1104_DIO12_OUT | DS1104_DIO13_OUT |
340                        DS1104_DIO14_OUT | DS1104_DIO15_OUT |
341                        DS1104_DIO16_OUT | DS1104_DIO17_OUT |
342                        DS1104_DIO18_OUT | DS1104_DIO19_OUT);
343
344     /* Loop that is performed whilst waiting for an interrupt */
345     while(1) {
346         while(msg_last_error_number() == DS1104_NO_ERROR) {
347
348             switch (main_loop_control) {
349             case 0: /* Flash the LEDs */
350                 /* This flashing routine runs every time led_flash_counter reaches 0xffff.  At this
351                  * point, the bits are shifted left with the addition of a new random bit. */
352
353                 led_flash_counter++;
```

```
354                 if (led_flash_counter == 0xffff) led_flash_counter = 0;
355                 if (led_flash_counter == 1) {
356                     led_flash_bits = (led_flash_bits << 1) + (rand() >> 8);
357                     ds1104_bit_io_write((led_flash_bits & 0xFFF00) + io_bits_write);
358                     if (led_flash_bits == 0)
359                         led_flash_bits = 0x1;
360                 }
361                 io_bits_status = ds1104_bit_io_read();
362                 break;
363             case 1: /* Read from the ADC and write to the DAC's */
364                 ds1104_adc_read_mux(adc_mux_scan_table, 4, adc_mux_values);
365                 ds1104_dac_write(1, adc_mux_values[2]);
366                 ds1104_dac_write(2, adc_mux_values[3]);
367                 break;
368             }
369             master_cmd_server();
370             host_service(0,0);
371         }
372
373         msg_info_set(MSG_SM_USER, msg_last_error_no,
374                      "Error occurred within PPC application 'adc_1104_hc.ppc'.");
375
376         while(msg_last_error_number() != DS1104_NO_ERROR) {
377             master_cmd_server();
378             host_service(0,0);
379         }
380         msg_info_set(MSG_SM_USER, 0, "Error released.");
381     }
382 }
```

A.2.1.2 Program description

This program is the C-program that is required to be compiled and installed on the dSPACE DS1104 controller board in order to run the experiment. The MATLAB functions used to communicate with the board are `RunDS1104-MIMO.m`, `RunDS1104Chkspk.m`, and `GetIRFsDS1104.m` that are presented in Section A.2.4. The code used for this experiment follows. The numbers in the left column refer to the line-numbers of the code.

Standard includes, and global variable definitions. (Lines 1-44)

1-9 General comment block used to describe the program.

11 *`Brtenv.h`* is the header file for the Basic Realtime Environment for the DS1104. It includes all the necessary header files, declaration of global variables, and definitions of macros [dSPACE, 2004].

14-52 These variables are defined globally (outside of any function) so that the dSPACE MATLAB MLIB function can obtain the address information and read the values from these variables.

The main function (Lines 324-382) The function, `main()`, defined at line 324 is the first function that is executed by the dSPACE board. This

function actually does nothing except to possibly (dependent on the value of the variable, `main_loop_control`) flash the LEDs on the break-out board, and also copy the input values to the output values to let the operator know that the program is loaded.

325 The variable, `adc_mux_scan_table`, contains an array of channel numbers to specify which channels to read from (see line 364).

326 The variable, `adc_mux_values`, is an array that is written to hold the read values from the A/D converter (see lines 364-366).

328 The `ds1104_init()` function is the standard DS1104 initialisation routine provided by dSPACE.

329 The `Initialise_Interrupt_Service()` function is defined at lines 310-321, and is used to specify which function to call when the host computer sends an interrupt.

330-342 These commands set up the D/A converter and the direction of the I/O bits on the device. All the bits are set as outputs to enable the program to "flash" the LEDs.

345-381 The program then operates in a single loop, awaiting an interrupt from the host computer.

364-371 This loop is executed whilst there is no error. If there is an error, lines 373-380 are used to handle the error, and after the error has been released, the loop from lines 345-381 is re-commenced.

348-368 Whilst the DS1104 board is waiting for an interrupt to occur, the device performs some functions depending on the value of the variable, `main_run_state`.

349-362 If the `main_run_state` variable is set to 0 then a continually incrementing value, `led_flash_number`, is written to the output, so that the LEDs appear to flash, (however only the high bits change slow enough for the eye to see the flashing). The bits are read into the variable iobits for retrieval from the host computer.

363-367 If the `main_run_state` variable is set to 1 then the A/D converter converters are read, and immediately written to the D/A converter outputs.

369-370 These two commands are used to allow MATLAB to communicate with the dSPACE board.

373-380 This is a set of error handling commands. This error handling routine was copied from the demonstration code provided by dSPACE.

The interrupt initialisation function (Lines 309-321) The interrupt initialisation function, `Initialise_Interrupt_Service()`, is called from the `main()` function on line 329.

312-314 This tells the interrupt handler to call `Host_Interrupt_Service()` when a host interrupt event occurs.

316-320 This enables the interrupts.

The interrupt service function (Lines 275-306) The interrupt service function, `Host_Interrupt_Service()`, is called when the host computer generates an interrupt on the card.

277,385 When the interrupt routine is running, the variable, `program_status,` is set to 1, and upon completion is set to 0.

278-304 The action to be taken as a result of the interrupt is determined by the value of the variable, `program_control`.

279-280 If the variable, `program_control`, is set to 1, nothing is executed.

281-290 If the variable, `program_control`, is set to 2, the function reads the A/D converters until a trigger level is recorded, and exits. (Such a function can allow the MATLAB script to wait for a loud noise to occur, before proceeding).

291-293 If the variable, `program_control`, is set to 3, the function `Play_Stepped_SIMO` is called.

294-296 If the variable, `program_control`, is set to 4, the function `Play_MIMO` is called.

297-298 If the variable, `program_control`, is set to 5, nothing is executed.

299-301 If the variable, `program_control`, is set to 5, the main_loop_control variable is set to 0.

The `Play_Stepped_SIMO` routine (Lines 55-164) The functions, `Play_Stepped_SIMO`, is used to play the same signal on each channel, and record the response on all the channels (a typical operation performed in the system identification procedure).

The `Play_MIMO` routine (Lines 167-271) The function, `Play_MIMO`, is used to simultaneously play and record sound at the same time similar to that of , `Play_Two_Out_In`, used in the first experiment.

A.2.2 DuctExperimentCreateTransmissionSignal.m

A.2.2.1 Program listing

```
% DuctExperimentCreateTransmissionSignal.m
%
% USAGE:
%           DuctExperimentCreateTransmissionSignal
%           DuctExperimentPlayAndPostProcess
%
% This script is used as part of an experiment conducted in a duct that forms part of the thesis for
% Pierre Dumuid. This script is designed to measure the system response and create a set of
% transmission according to various inverse filter designs. The script,
% DuctExperimentPlayAndPostProcess, is then used to play the signals in the environment and
% post-process the signals to display various performance metrics.
%
% Before invoking this M-file the real-time processor application, XXXX.obj
% (or equivalent) must be loaded and running on the dSPACE board.

% These variables hold the filter designs for:
% No Filtering, Time Reversal, Milica's 2-sided, IF:Path, IF:Channel, IF:Full-MIMO
clear nof tr mil2s if_path if_chan if_fmimo;

%=========================================================================
% Define experiment parameters
%=========================================================================

freqSampling = 57000;
freqCarrier  = 4000;
symbolLength = 25;

playChannels   = [1 2 3 4 5 6];
recordChannels = [1:2];

channelSettings.playArrayId   = 1;
channelSettings.recordArrayId = 2;

config.loadMatChannel              = 1;
config.simulateChannel             = 1;
config.downSampleForFilterDesign = 1;
config.receiverFilter              = 1;

config.matChannelFilename = '/home/pmdumuid/repofind/LastNonGit/testIRFs.mat';

%=========================================================================
% Measure Impulse Response Functions (IRFs)
%=========================================================================
fprintf('Measuring channelIRFs\n');
if (config.loadMatChannel)
    a = load(config.matChannelFilename);
    channelIRFs = a.IRFs(1:1024,:,:);
elseif (config.simulateChannel)
    channelIRFs = GetFakeIRFs(length(playChannels),length(recordChannels),1024);
    noise = rand(6000,length(playChannels),length(recordChannels));
else
    a = load('micgains'); smgains.mic = a.micgains.current;
    b = load('spkgains'); smgains.spk = b.spkgains.current;
    if channelSettings.playArrayId   == 1
        channelSettings.playGains = smgains.spk.tx1;
    else
        channelSettings.playGains = smgains.spk.tx2;
    end
    if channelSettings.recordArrayId == 1
        channelSettings.playGains = smgains.spk.rx1;
    else
```

```
62          channelSettings.playGains = smgains.spk.rx2;
63      end
64      [channelIRFs coherence noise] = GetIRFsDS1104(freqSampling,playChannels,recordChannels, ...
65                                                    channelSettings);
66  end
67
68  %=========================================================================
69  % Convert IRF's to baseband
70  %=========================================================================
71  fprintf('Converting to baseband\n');
72  % Truncate IRFs to 2048.
73  channelIRFLength = min(2048,size(channelIRFs,1));
74
75  channelIRFsBB = PassbandToBaseband(channelIRFs(1:channelIRFLength,:,:),freqSampling,freqCarrier);
76
77  %=========================================================================
78  % Down-sample the filters
79  %=========================================================================
80  fprintf('Resample to baseband\n');
81  if (config.downSampleForFilterDesign)
82      % Set baseband to min(3.5 time samples per symbol, freqCarrier)
83      freqSamplingBB = min(floor(3.5*freqSampling/symbolLength),freqCarrier);
84
85      channelIRFsBBL = ResampleIRFs(channelIRFsBB,freqSamplingBB,freqSampling);
86  else
87      channelIRFsBBL = channelIRFsBB;
88  end
89
90  %=========================================================================
91  % Design the inverse filters
92  %=========================================================================
93  fprintf('Design of baseband filters\n');
94
95  nof.inverseIRFsBBL      = CreateInverseFilter(channelIRFsBBL,'No Filtering');
96
97  tr.inverseIRFsBBL       = CreateInverseFilter(channelIRFsBBL,'TimeReversal');
98
99  filterSettings.mil2s.beta = 0;
100 [mil2s.inverseIRFsBBL mil2s.extraInfo] = CreateInverseFilter(channelIRFsBBL, ...
101                                                              'Milica: Two-Side Filter', ...
102                                                              filterSettings.mil2s);
103
104 filterSettings.if_path.beta = 0.0005;
105 if_path.inverseIRFsBBL  = CreateInverseFilter(channelIRFsBBL, ...
106                                               'Tikhonov IF: Path', ...
107                                               filterSettings.if_path);
108
109 filterSettings.if_chan.beta = 0.0005;
110 if_chan.inverseIRFsBBL  = CreateInverseFilter(channelIRFsBBL, ...
111                                               'Tikhonov IF: Channel', ...
112                                               filterSettings.if_chan);
113
114 filterSettings.if_fmimo.beta = .0005;
115 if_fmimo.inverseIRFsBBL = CreateInverseFilter(channelIRFsBBL, ...
116                                               'Tikhonov IF: Full MIMO', ...
117                                               filterSettings.if_fmimo);
118
119 %=========================================================================
120 % Up-sample filter
121 %=========================================================================
122 fprintf('Resample the channelIRFs from freqSamplingBB to freqSampling\n');
123 if (config.downSampleForFilterDesign)
124     nof.inverseIRFsBB       = ResampleIRFs(nof.inverseIRFsBBL       ,freqSampling,freqSamplingBB);
125     tr.inverseIRFsBB        = ResampleIRFs(tr.inverseIRFsBBL        ,freqSampling,freqSamplingBB);
126     mil2s.inverseIRFsBB     = ResampleIRFs(mil2s.inverseIRFsBBL     ,freqSampling,freqSamplingBB);
```

```
127     if_path.inverseIRFsBB  = ResampleIRFs(if_path.inverseIRFsBBL  ,freqSampling,freqSamplingBB);
128     if_chan.inverseIRFsBB  = ResampleIRFs(if_chan.inverseIRFsBBL  ,freqSampling,freqSamplingBB);
129     if_fmimo.inverseIRFsBB = ResampleIRFs(if_fmimo.inverseIRFsBBL,freqSampling,freqSamplingBB);
130 else
131     nof.inverseIRFsBB      = nof.inverseIRFsBBL;
132     tr.inverseIRFsBB       = tr.inverseIRFsBBL;
133     mil2s.inverseIRFsBB    = mil2s.inverseIRFsBBL;
134     if_path.inverseIRFsBB  = if_path.inverseIRFsBBL;
135     if_chan.inverseIRFsBB  = if_chan.inverseIRFsBBL;
136     if_fmimo.inverseIRFsBB = if_fmimo.inverseIRFsBBL;
137 end
138
139 %=========================================================================
140 % Create the transmission signal
141 %=========================================================================
142 fprintf('Make up a transmission signal\n');
143
144 fprintf('(a) Create a random binary stream\n');
145 digitalSequence = [1 [rand(1,499) > 0.5]];
146
147 fprintf('(b) Map binary stream to complex symbols (PSK)\n');
148 modulationSettings.k         = 2;
149 modulationSettings.addphase = 0;
150 complexSymbols = BitSequenceToComplexSequence(digitalSequence,'PSK',modulationSettings);
151
152 fprintf('(c) Prepend symbols with a lead in sequence (used for synchronisation)\n');
153 % The lead-in sequence is used to synchronise the decoder, and perform a one-time phase correction.
154 leadInSequence = [1 zeros(1,5) 1 zeros(1,5)];
155 complexSymbols = [leadInSequence complexSymbols];
156
157 fprintf('(d) Create the spectral shaping filters\n');
158 rCFLength = 2048;
159 rCFLength = ceil(rCFLength/symbolLength)*symbolLength;
160
161 % Create the raised co-sine spectral shaping filters.
162 raisedCosineSqrtFilter = real(fftshift(ifft(sqrt( ...
163     RaisedCosineFrequencySpectrum([1:rCFLength/2 -rCFLength/2+1:0]*freqSampling/rCFLength, ...
164                                   symbolLength/freqSampling, ...
165                                   0.2) ...
166     ))));
167 raisedCosineFilter     = real(fftshift(ifft( ...
168     RaisedCosineFrequencySpectrum([1:rCFLength/2 -rCFLength/2+1:0]*freqSampling/rCFLength, ...
169                                   symbolLength/freqSampling, ...
170                                   0.2) ...
171     )));
172
173 % Normalise the filter co-efficients
174 raisedCosineFilter     = raisedCosineFilter     /sum(raisedCosineFilter);
175 raisedCosineSqrtFilter = raisedCosineSqrtFilter/sum(raisedCosineSqrtFilter);
176
177 fprintf('(e) Convert the symbol stream into the base-band stream.\n');
178 transmitSignalBB = ComplexSequenceToSignal(complexSymbols, ...
179                                            raisedCosineSqrtFilter,symbolLength);
180
181 % Second channel is not transmitted.
182 transmitSignalBB(1:size(transmitSignalBB,1),2) ...
183     = zeros(1,size(transmitSignalBB,1));
184
185 %=========================================================================
186 % Apply inverse filtering to transmission signals
187 %=========================================================================
188 fprintf('Convolving with the TX signal with the filters \n');
189 fprintf('          --> Base-band Filtering for No Filtering\n');
190 transmitSignalBBIF.nof       = MatrixConvolve(nof.inverseIRFsBB       ,transmitSignalBB);
191 fprintf('          --> Base-band Filtering for TR\n');
```

```
transmitSignalBBIF.tr        = MatrixConvolve(tr.inverseIRFsBB        ,transmitSignalBB);
fprintf('          --> Base-band Filtering for Mil 2s\n');
transmitSignalBBIF.mil2s     = MatrixConvolve(mil2s.inverseIRFsBB     ,transmitSignalBB);
fprintf('          --> Base-band Filtering for if_path\n');
transmitSignalBBIF.if_path   = MatrixConvolve(if_path.inverseIRFsBB   ,transmitSignalBB);
fprintf('          --> Base-band Filtering for if_chan\n');
transmitSignalBBIF.if_chan   = MatrixConvolve(if_chan.inverseIRFsBB   ,transmitSignalBB);
fprintf('          --> Base-band Filtering for if_fmimo\n');
transmitSignalBBIF.if_fmimo = MatrixConvolve(if_fmimo.inverseIRFsBB,transmitSignalBB);

%=========================================================================
% Convert signals to passband
%=========================================================================
fprintf('STEP 6: Convert the signals to pass-band\n');
txSignalInfo.PSK.transmitSignalIF.nof = ...
    BasebandToPassband(transmitSignalBBIF.nof       ,freqCarrier,freqSampling);
txSignalInfo.PSK.transmitSignalIF.tr = ...
    BasebandToPassband(transmitSignalBBIF.tr        ,freqCarrier,freqSampling);
txSignalInfo.PSK.transmitSignalIF.mil2s = ...
    BasebandToPassband(transmitSignalBBIF.mil2s     ,freqCarrier,freqSampling);
txSignalInfo.PSK.transmitSignalIF.if_path = ...
    BasebandToPassband(transmitSignalBBIF.if_path   ,freqCarrier,freqSampling);
txSignalInfo.PSK.transmitSignalIF.if_chan = ...
    BasebandToPassband(transmitSignalBBIF.if_chan   ,freqCarrier,freqSampling);
txSignalInfo.PSK.transmitSignalIF.if_fmimo = ...
    BasebandToPassband(transmitSignalBBIF.if_fmimo,freqCarrier,freqSampling);

%=========================================================================
% Normalise the signals
%=========================================================================
fprintf('STEP 7: Normalisations:\n');
txSignalInfo.PSK.transmitSignalIF.nof = ...
    txSignalInfo.PSK.transmitSignalIF.nof       /max(txSignalInfo.PSK.transmitSignalIF.nof(:));
txSignalInfo.PSK.transmitSignalIF.tr = ...
    txSignalInfo.PSK.transmitSignalIF.tr        /max(txSignalInfo.PSK.transmitSignalIF.tr(:));
txSignalInfo.PSK.transmitSignalIF.mil2s = ...
    txSignalInfo.PSK.transmitSignalIF.mil2s     /max(txSignalInfo.PSK.transmitSignalIF.mil2s(:));
txSignalInfo.PSK.transmitSignalIF.if_path = ...
    txSignalInfo.PSK.transmitSignalIF.if_path   /max(txSignalInfo.PSK.transmitSignalIF.if_path(:));
txSignalInfo.PSK.transmitSignalIF.if_chan = ...
    txSignalInfo.PSK.transmitSignalIF.if_chan   /max(txSignalInfo.PSK.transmitSignalIF.if_chan(:));
txSignalInfo.PSK.transmitSignalIF.if_fmimo = ...
    txSignalInfo.PSK.transmitSignalIF.if_fmimo/max(txSignalInfo.PSK.transmitSignalIF.if_fmimo(:));

%=========================================================================
% Retain some parameters
%=========================================================================
txSignalInfo.PSK.leadInSequence       = leadInSequence;
txSignalInfo.PSK.leadInSequenceLength = length(leadInSequence);
txSignalInfo.PSK.modulationSettings   = modulationSettings;
txSignalInfo.PSK.complexSymbols       = complexSymbols;
```

A.2.3 DuctExperimentPlayAndPostprocess.m

A.2.3.1 Program listing

```
% DuctExperimentPlayAndPostProcess.m
%
% USAGE:
%           DuctExperimentCreateTransmissionSignal
%           DuctExperimentPlayAndPostProcess
%
```

```
% This script is used as part of an experiment conducted in a duct that forms part of the thesis for
% Pierre Dumuid. This script is to be run after the script, DuctExperimentCreateTransmissionSignal
% has been run. The script, DuctExperimentCreateTransmissionSignal, is designed to measure the
% system response and create a set of transmission according to inverse filter designs. This script
% is then used to play the signals in the environment and post-process the signals to display
% various performance metrics. The postprocessing involves:
%
%     R Plotting the signal received at the target and non-target location
%     P Performing a phase and signal synchronisation based on the first peak
%     P Converting the signal to base-band
%     R Displaying a scatter plot of the signal at the target and non-target locations
%     P Pass the signal through the ZF, MSE and RLS adaptive algorithms and show the following:
%        R Scatter plot after adaptive filtering
%        R Plot of the error for each step
%        R Filter tap history (to observe how the taps change as the filter adapts)
%
% where P stands for post-processing, and R stand for showing results.
%

config.showReceivedSignal = 0;

%=========================================================================
% Specify parameters for the adaptive filters
%=========================================================================
adaptiveFIR.deltaSteps    = [0   1e-2  3e-3  1e-4  1e-4  1e-5];
adaptiveFIR.deltaStepsFS  = [0   6e-3  4e-4  4e-4  1e-4  1e-5];
adaptiveFIR.deltaStepsRLS = [0 0.999 0.999 0.999 0.999 0.999];
% filterPeakIndex specifies where the peak is for the RLS and MSE adaptive algorithmes. Values range
% between 1 and filterLength. A value of 1 indicates that the filter operates on the current and
% past sampled signals only, and a value of 15 indicates the filter operates on the current and
% future sampled signals.
adaptiveFIR.filterLength          = 15;
adaptiveFIR.filterPeakIndex       = 15;
adaptiveFIR.feedbackFilterLength = 48;
adaptiveFIR.fractionalSteps       = 2;

% Define some useful variables
leadInSequenceLength = txSignalInfo.PSK.leadInSequenceLength;
complexSymbolsLength = length(txSignalInfo.PSK.complexSymbols);

% Make the leadInSequence and the first 50 symbols to be the training sequence.
trainingSequence = txSignalInfo.PSK.complexSymbols(leadInSequenceLength + [1:50]);

%=========================================================================
% Obtain the system response from playing the signal.
%=========================================================================
if (config.simulateChannel)
    fprintf('Calculating response for No Filtering\n');
    txSignalInfo.PSK.receivedSignal.nof = ...
        MatrixConvolve(channelIRFs,txSignalInfo.PSK.transmitSignalIF.nof);
    fprintf('Calculating response for tr\n');
    txSignalInfo.PSK.receivedSignal.tr = ...
        MatrixConvolve(channelIRFs,txSignalInfo.PSK.transmitSignalIF.tr);
    fprintf('Calculating response for mil2s\n');
    txSignalInfo.PSK.receivedSignal.mil2s = ...
        MatrixConvolve(channelIRFs,txSignalInfo.PSK.transmitSignalIF.mil2s);
    fprintf('Calculating response for if_path\n');
    txSignalInfo.PSK.receivedSignal.if_path = ...
        MatrixConvolve(channelIRFs,txSignalInfo.PSK.transmitSignalIF.if_path);
    fprintf('Calculating response for if_chan\n');
    txSignalInfo.PSK.receivedSignal.if_chan = ...
        MatrixConvolve(channelIRFs,txSignalInfo.PSK.transmitSignalIF.if_chan);
    fprintf('Calculating response for if_fmimo\n');
    txSignalInfo.PSK.receivedSignal.if_fmimo = ...
        MatrixConvolve(channelIRFs,txSignalInfo.PSK.transmitSignalIF.if_fmimo);
```

```
72  else
73      % Set the deviceSetting structure
74      deviceSettings.freqSampling          = freqSampling;
75      deviceSettings.playChannels          = playChannels;
76      deviceSettings.recordChannels        = recordChannels;
77      deviceSettings.playChannelsCount     = length(deviceSettings.playChannels);
78      deviceSettings.recordChannelsCount = length(deviceSettings.recordChannels);
79
80      fprintf('Measuring response for nof\n');
81      transmitSignal = txSignalInfo.PSK.transmitSignalIF.nof;
82      deviceSettings.playSampleLength = size(transmitSignal,1);
83      deviceSettings.recordSampleLength = size(transmitSignal,1) + channelIRFLength;
84      txSignalInfo.PSK.receivedSignal.nof = RunDS1104MIMO(deviceSettings,transmitSignal);
85
86      fprintf('Measuring response for tr\n');
87      transmitSignal = txSignalInfo.PSK.transmitSignalIF.tr;
88      deviceSettings.playSampleLength = size(transmitSignal,1);
89      deviceSettings.recordSampleLength = size(transmitSignal,1) + channelIRFLength;
90      txSignalInfo.PSK.receivedSignal.tr = RunDS1104MIMO(deviceSettings,transmitSignal);
91
92      fprintf('Measuring response for Mil 2s\n');
93      transmitSignal = txSignalInfo.PSK.transmitSignalIF.mil2s;
94      deviceSettings.playSampleLength = size(transmitSignal,1);
95      deviceSettings.recordSampleLength = size(transmitSignal,1) + channelIRFLength;
96      txSignalInfo.PSK.receivedSignal.mil2s = RunDS1104MIMO(deviceSettings,transmitSignal);
97
98      fprintf('Measuring response for if_path\n');
99      transmitSignal = txSignalInfo.PSK.transmitSignalIF.if_path;
100     deviceSettings.playSampleLength = size(transmitSignal,1);
101     deviceSettings.recordSampleLength = size(transmitSignal,1) + channelIRFLength;
102     txSignalInfo.PSK.receivedSignal.if_path = RunDS1104MIMO(deviceSettings,transmitSignal);
103
104     fprintf('Measuring response for if_chan\n');
105     transmitSignal = txSignalInfo.PSK.transmitSignalIF.if_chan;
106     deviceSettings.playSampleLength = size(transmitSignal,1);
107     deviceSettings.recordSampleLength = size(transmitSignal,1) + channelIRFLength;
108     txSignalInfo.PSK.receivedSignal.if_chan = RunDS1104MIMO(deviceSettings,transmitSignal);
109
110     fprintf('Measuring response for if_fmimo\n');
111     transmitSignal = txSignalInfo.PSK.transmitSignalIF.if_fmimo;
112     deviceSettings.playSampleLength = size(transmitSignal,1);
113     deviceSettings.recordSampleLength = size(transmitSignal,1) + channelIRFLength;
114     txSignalInfo.PSK.receivedSignal.if_fmimo = RunDS1104MIMO(deviceSettings,transmitSignal);
115 end
116
117 %=========================================================================
118 % Calculate delay if using a receiver filter
119 %=========================================================================
120 if (config.receiverFilter)
121     [a rxFilterDelay] = max(raisedCosineSqrtFilter);
122 else
123     rxFilterDelay = 0;
124 end
125
126 % Design a low pass filter for the SignalPhaseEstimatorPassbandToBaseband routine
127 lowPassFIR = fir1(128,freqCarrier/freqSampling);
128
129 figureCount = 0;
130
131 % Initialise parameters for figures that show results for each inverse filter on multiple rows.
132 multiRowFigureDetails.totalRows  = 5;
133 multiRowFigureDetails.currentRow = 0;
134
135 %=========================================================================
136 % Initialise the figure windows for each of the multi-row figures.
```

```
%=========================================================================
figureCount = figureCount + 1;
multiRowFigureDetails.figureHandles.scatterPlots = figure(figureCount);
set(multiRowFigureDetails.figureHandles.scatterPlots,'Position',[5    230    310    718]);
figureCount = figureCount + 1;
multiRowFigureDetails.figureHandles.adaptiveScatterPlots = figure(figureCount);
set(multiRowFigureDetails.figureHandles.adaptiveScatterPlots,'Position',[325    227    547    718]);
figureCount = figureCount + 1;
multiRowFigureDetails.figureHandles.symbolErrorHistory = figure(figureCount);
set(multiRowFigureDetails.figureHandles.symbolErrorHistory,'Position',[880    227    547    718]);

%=========================================================================
% 1. Process the results for Milica Stojanovic 2-sided filter
%=========================================================================
currentFilterTextDescription  = 'Stoj. 2-S';
multiRowFigureDetails.currentRow = multiRowFigureDetails.currentRow + 1;
if (config.showReceivedSignal)
    figureCount = figureCount + 1;
    figure(figureCount);
    set(figureCount,'Name',currentFilterTextDescription);
    subplot(2,1,1);
    plot(txSignalInfo.PSK.receivedSignal.mil2s(:,1));
    title(['Received signals for ' currentFilterTextDescription]);
    subplot(2,1,2);
    plot(txSignalInfo.PSK.receivedSignal.mil2s(:,2));
    set(get(gcf,'children'),'YLim',[-1 1]*max(max(abs(txSignalInfo.PSK.receivedSignal.mil2s))));
end

%-------------------------------------------------------------------------
% a. Convert the signals of both channels to baseband, and apply the receiver filter as appropriate.
%-------------------------------------------------------------------------
[receivedSignalBB1 maxValIdx1 phiPeak gain1] = ...
    SignalPhaseEstimatorPassbandToBaseband(txSignalInfo.PSK.receivedSignal.mil2s(:,1), ...
                                           freqCarrier,freqSampling,lowPassFIR,symbolLength);

[receivedSignalBB2 maxValIdx2 phiPeak gain2] = ...
    SignalPhaseEstimatorPassbandToBaseband(txSignalInfo.PSK.receivedSignal.mil2s(:,2), ...
                                           freqCarrier,freqSampling,lowPassFIR,symbolLength);

fprintf([currentFilterTextDescription ' maxValIdx1 = %d\n'],maxValIdx1);

if (config.receiverFilter)
    receivedSignalBB1 = conv(raisedCosineSqrtFilter,receivedSignalBB1);
    receivedSignalBB2 = conv(raisedCosineSqrtFilter,receivedSignalBB2);
end

% The two-sided filter design by Milica requires the received signal be filtered by another
% filter developed by the algorithm.
receivedSignalBB1 = conv(receivedSignalBB1,mil2s.extraInfo.rxFilter(:,1));
milicaReceiverFilterDelay = rxFilterDelay + size(mil2s.extraInfo.rxFilter,1)/2;

PlotScatter( ...
    receivedSignalBB1( ...
        (maxValIdx1 + milicaReceiverFilterDelay) ...
        + [leadInSequenceLength*symbolLength:complexSymbolsLength*symbolLength])/gain1, ...
    receivedSignalBB2( ...
        (maxValIdx1+milicaReceiverFilterDelay) ...
        + [leadInSequenceLength*symbolLength:complexSymbolsLength*symbolLength])/gain2, ...
    symbolLength,multiRowFigureDetails,currentFilterTextDescription);

%-------------------------------------------------------------------------
% b. Re-sample at symbol locations (and at the fractional spacing rate)
%-------------------------------------------------------------------------
sampledSignal    = receivedSignalBB1(maxValIdx1 + milicaReceiverFilterDelay ...
                                     + [ ...
```

```
                                        leadInSequenceLength*symbolLength ...
                                        :symbolLength ...
                                        :complexSymbolsLength*symbolLength]) ';

sampledSignalFS = receivedSignalBB1(maxValIdx1 + milicaReceiverFilterDelay ...
                                    + round([ ...
                                        leadInSequenceLength*symbolLength ...
                                        :symbolLength/adaptiveFIR.fractionalSteps ...
                                        :complexSymbolsLength*symbolLength])) ';

%-------------------------------------------------------------------------------
% c. Pass the signal through the adaptive algorithms
%-------------------------------------------------------------------------------
figureCount = figureCount + 1;
figure(figureCount);
set(figureCount,'Name',currentFilterTextDescription, ...
                'Position',[664   178   454   592]);
constellationValues = GetConstellationValues('PSK',0,txSignalInfo.PSK.modulationSettings);
adaptiveFilterData.mil2s = ...
    DuctExperimentRunAdaptiveTests(constellationValues,sampledSignal,sampledSignalFS, ...
                                   trainingSequence,multiRowFigureDetails, ...
                                   currentFilterTextDescription,adaptiveFIR);

%===============================================================================
% 2. Process the results for the Time Reversal filter
%===============================================================================
currentFilterTextDescription='T.R.';
multiRowFigureDetails.currentRow = multiRowFigureDetails.currentRow + 1;
if (config.showReceivedSignal)
    figureCount = figureCount + 1;
    figure(figureCount);
    set(figureCount,'Name',currentFilterTextDescription);
    subplot(2,1,1);
    plot(txSignalInfo.PSK.receivedSignal.tr(:,1));
    title(['Received signals for ' currentFilterTextDescription]);
    subplot(2,1,2);
    plot(txSignalInfo.PSK.receivedSignal.tr(:,2));
    set(get(gcf,'children'),'YLim',[-1 1]*max(max(abs(txSignalInfo.PSK.receivedSignal.tr))));
end

%-------------------------------------------------------------------------------
% a. Convert the signals of both channels to baseband, and apply the receiver filter as appropriate.
%-------------------------------------------------------------------------------
[receivedSignalBB1 maxValIdx1 phiPeak gain1] = ...
    SignalPhaseEstimatorPassbandToBaseband(txSignalInfo.PSK.receivedSignal.tr(:,1),freqCarrier, ...
                                           freqSampling,lowPassFIR,symbolLength);

[receivedSignalBB2 maxValIdx2 phiPeak gain2] = ...
    SignalPhaseEstimatorPassbandToBaseband(txSignalInfo.PSK.receivedSignal.tr(:,2),freqCarrier, ...
                                           freqSampling,lowPassFIR,symbolLength);

fprintf([currentFilterTextDescription ' maxValIdx1 = %d\n'],maxValIdx1);

if (config.receiverFilter)
    receivedSignalBB1 = conv(raisedCosineSqrtFilter,receivedSignalBB1);
    receivedSignalBB2 = conv(raisedCosineSqrtFilter,receivedSignalBB2);
end

PlotScatter( ...
    receivedSignalBB1( ...
        (maxValIdx1+rxFilterDelay) ...
        + [leadInSequenceLength*symbolLength:complexSymbolsLength*symbolLength])/gain1, ...
    receivedSignalBB2( ...
        (maxValIdx1+rxFilterDelay) ...
        + [leadInSequenceLength*symbolLength:complexSymbolsLength*symbolLength])/gain2, ...
```

```
267     symbolLength, multiRowFigureDetails,currentFilterTextDescription);
268
269 %-------------------------------------------------------------------------
270 % b. Re-sample at symbol locations (and at the fractional spacing rate)
271 %-------------------------------------------------------------------------
272 sampledSignal    = receivedSignalBB1(maxValIdx1 + rxFilterDelay ...
273                                      + [ ...
274                                          leadInSequenceLength*symbolLength ...
275                                          :symbolLength ...
276                                          :complexSymbolsLength*symbolLength])';
277
278 sampledSignalFS = receivedSignalBB1(maxValIdx1 + rxFilterDelay ...
279                                      + round([ ...
280                                          leadInSequenceLength*symbolLength ...
281                                          :symbolLength/adaptiveFIR.fractionalSteps ...
282                                          :complexSymbolsLength*symbolLength]))';
283
284 %-------------------------------------------------------------------------
285 % c. Pass the signal through the adaptive algorithms
286 %-------------------------------------------------------------------------
287 figureCount = figureCount + 1;
288 figure(figureCount);
289 set(figureCount,'Name',currentFilterTextDescription, ...
290                 'Position',[664   178   454   592]);
291 constellationValues = GetConstellationValues('PSK',0,txSignalInfo.PSK.modulationSettings);
292 adaptiveFilterData.tr =  ...
293     DuctExperimentRunAdaptiveTests(constellationValues,sampledSignal,sampledSignalFS, ...
294                                    trainingSequence,multiRowFigureDetails, ...
295                                    currentFilterTextDescription,adaptiveFIR);
296
297 %=========================================================================
298 % 3. Process the results for Tikhonov inverse filter (full)
299 %=========================================================================
300 currentFilterTextDescription='T.I.F.(full)';
301 multiRowFigureDetails.currentRow = multiRowFigureDetails.currentRow + 1;
302 if (config.showReceivedSignal)
303     figureCount = figureCount + 1;
304     figure(figureCount);
305     set(figureCount,'Name',currentFilterTextDescription);
306     subplot(2,1,1);
307     plot(txSignalInfo.PSK.receivedSignal.if_fmimo(:,1));
308     title(['Received signals for ' currentFilterTextDescription]);
309     subplot(2,1,2);
310     plot(txSignalInfo.PSK.receivedSignal.if_fmimo(:,2));
311     set(get(gcf,'children'),'YLim',[-1 1]*max(max(abs(txSignalInfo.PSK.receivedSignal.if_fmimo))));
312 end
313
314 %-------------------------------------------------------------------------
315 % a. Convert the signals of both channels to baseband, and apply the receiver filter as appropriate.
316 %-------------------------------------------------------------------------
317 [receivedSignalBB1 maxValIdx1 phiPeak gain1] = ...
318     SignalPhaseEstimatorPassbandToBaseband(txSignalInfo.PSK.receivedSignal.if_fmimo(:,1),freqCarrier, ...
319                                            freqSampling,lowPassFIR,symbolLength);
320
321 [receivedSignalBB2 maxValIdx2 phiPeak gain2] = ...
322     SignalPhaseEstimatorPassbandToBaseband(txSignalInfo.PSK.receivedSignal.if_fmimo(:,2),freqCarrier, ...
323                                            freqSampling,lowPassFIR,symbolLength);
324
325 fprintf([currentFilterTextDescription ' maxValIdx1 = %d\n'],maxValIdx1);
326
327 if (config.receiverFilter)
328     receivedSignalBB1 = conv(raisedCosineSqrtFilter,receivedSignalBB1);
329     receivedSignalBB2 = conv(raisedCosineSqrtFilter,receivedSignalBB2);
330 end
331
```

```
PlotScatter( ...
    receivedSignalBB1( ...
        (maxValIdx1+rxFilterDelay) ...
        + [leadInSequenceLength*symbolLength:complexSymbolsLength*symbolLength])/gain1, ...
    receivedSignalBB2( ...
        (maxValIdx1+rxFilterDelay) ...
        + [leadInSequenceLength*symbolLength:complexSymbolsLength*symbolLength])/gain2, ...
    symbolLength, multiRowFigureDetails,currentFilterTextDescription);

%-------------------------------------------------------------------------------------------
% b. Re-sample at symbol locations (and at the fractional spacing rate)
%-------------------------------------------------------------------------------------------
sampledSignal    = receivedSignalBB1(maxValIdx1 + rxFilterDelay ...
                                     + [ ...
                                         leadInSequenceLength*symbolLength ...
                                         :symbolLength ...
                                         :complexSymbolsLength*symbolLength])';

sampledSignalFS = receivedSignalBB1(maxValIdx1 + rxFilterDelay ...
                                     + round([ ...
                                         leadInSequenceLength*symbolLength ...
                                         :symbolLength/adaptiveFIR.fractionalSteps ...
                                         :complexSymbolsLength*symbolLength]))';

%-------------------------------------------------------------------------------------------
% c. Pass the signal through the adaptive algorithms
%-------------------------------------------------------------------------------------------
figureCount = figureCount + 1;
figure(figureCount);
set(figureCount,'Name',currentFilterTextDescription, ...
                'Position',[664    178    454    592]);
constellationValues = GetConstellationValues('PSK',0,txSignalInfo.PSK.modulationSettings);
adaptiveFilterData.tiffull =  ...
    DuctExperimentRunAdaptiveTests(constellationValues,sampledSignal,sampledSignalFS, ...
                                   trainingSequence,multiRowFigureDetails, ...
                                   currentFilterTextDescription,adaptiveFIR);

%===========================================================================================
% 4. Process the results for Tikhonov inverse filter (one channel)
%===========================================================================================
currentFilterTextDescription='T.I.F.(channel)';
multiRowFigureDetails.currentRow = multiRowFigureDetails.currentRow + 1;
if (config.showReceivedSignal)
    figureCount = figureCount + 1;
    figure(figureCount);
    set(figureCount,'Name',currentFilterTextDescription);
    subplot(2,1,1);
    plot(txSignalInfo.PSK.receivedSignal.if_chan(:,1));
    title(['Received signals for ' currentFilterTextDescription]);
    subplot(2,1,2);
    plot(txSignalInfo.PSK.receivedSignal.if_chan(:,2));
    set(get(gcf,'children'),'YLim',[-1 1]*max(max(abs(txSignalInfo.PSK.receivedSignal.if_chan))));
end

%-------------------------------------------------------------------------------------------
% a. Convert the signals of both channels to baseband, and apply the receiver filter as appropriate.
%-------------------------------------------------------------------------------------------
[receivedSignalBB1 maxValIdx1 phiPeak gain1] = ...
    SignalPhaseEstimatorPassbandToBaseband(txSignalInfo.PSK.receivedSignal.if_chan(:,1),freqCarrier, ...
                                           freqSampling,lowPassFIR,symbolLength);

[receivedSignalBB2 maxValIdx2 phiPeak gain2] = ...
    SignalPhaseEstimatorPassbandToBaseband(txSignalInfo.PSK.receivedSignal.if_chan(:,2),freqCarrier, ...
                                           freqSampling,lowPassFIR,symbolLength);

```

```
397 fprintf([currentFilterTextDescription ' maxValIdx1 = %d\n'],maxValIdx1);
398
399 if (config.receiverFilter)
400     receivedSignalBB1 = conv(raisedCosineSqrtFilter,receivedSignalBB1);
401     receivedSignalBB2 = conv(raisedCosineSqrtFilter,receivedSignalBB2);
402 end
403
404 PlotScatter( ...
405     receivedSignalBB1( ...
406         (maxValIdx1+rxFilterDelay) ...
407         + [leadInSequenceLength*symbolLength:complexSymbolsLength*symbolLength])/gain1, ...
408     receivedSignalBB2( ...
409         (maxValIdx1+rxFilterDelay) ...
410         + [leadInSequenceLength*symbolLength:complexSymbolsLength*symbolLength])/gain2, ...
411     symbolLength, multiRowFigureDetails,currentFilterTextDescription);
412
413 %-------------------------------------------------------------------------
414 % b. Re-sample at symbol locations (and at the fractional spacing rate)
415 %-------------------------------------------------------------------------
416 sampledSignal   = receivedSignalBB1(maxValIdx1 + rxFilterDelay ...
417                                     + [ ...
418                                         leadInSequenceLength*symbolLength ...
419                                         :symbolLength ...
420                                         :complexSymbolsLength*symbolLength])';
421
422 sampledSignalFS = receivedSignalBB1(maxValIdx1 + rxFilterDelay ...
423                                     + round([ ...
424                                         leadInSequenceLength*symbolLength ...
425                                         :symbolLength/adaptiveFIR.fractionalSteps ...
426                                         :complexSymbolsLength*symbolLength]))';
427
428 %-------------------------------------------------------------------------
429 % c. Pass the signal through the adaptive algorithms
430 %-------------------------------------------------------------------------
431 figureCount = figureCount + 1;
432 figure(figureCount);
433 set(figureCount,'Name',currentFilterTextDescription, ...
434                 'Position',[664   178   454   592]);
435 constellationValues = GetConstellationValues('PSK',0,txSignalInfo.PSK.modulationSettings);
436 adaptiveFilterData.tifchan =  ...
437     DuctExperimentRunAdaptiveTests(constellationValues,sampledSignal,sampledSignalFS, ...
438                                    trainingSequence,multiRowFigureDetails, ...
439                                    currentFilterTextDescription,adaptiveFIR);
440
441 %=========================================================================
442 % 5. Process the results for Tikhonov inverse filter (one path)
443 %=========================================================================
444 currentFilterTextDescription='T.I.F.(path)';
445 multiRowFigureDetails.currentRow = multiRowFigureDetails.currentRow + 1;
446 if (config.showReceivedSignal)
447     figureCount = figureCount + 1;
448     figure(figureCount);
449     set(figureCount,'Name',currentFilterTextDescription);
450     subplot(2,1,1);
451     plot(txSignalInfo.PSK.receivedSignal.if_path(:,1));
452     title(['Received signals for ' currentFilterTextDescription]);
453     subplot(2,1,2);
454     plot(txSignalInfo.PSK.receivedSignal.if_path(:,2));
455     set(get(gcf,'children'),'YLim',[-1 1]*max(max(abs(txSignalInfo.PSK.receivedSignal.if_path))));
456 end
457
458 %-------------------------------------------------------------------------
459 % a. Convert the signals of both channels to baseband, and apply the receiver filter as appropriate.
460 %-------------------------------------------------------------------------
461 [receivedSignalBB1 maxValIdx1 phiPeak gain1] = ...
```

```
462     SignalPhaseEstimatorPassbandToBaseband(txSignalInfo.PSK.receivedSignal.if_path(:,1), ...
463                                            freqCarrier,freqSampling,lowPassFIR,symbolLength);
464
465 [receivedSignalBB2 maxValIdx2 phiPeak gain2] = ...
466     SignalPhaseEstimatorPassbandToBaseband(txSignalInfo.PSK.receivedSignal.if_path(:,2), ...
467                                            freqCarrier,freqSampling,lowPassFIR,symbolLength);
468
469 fprintf([currentFilterTextDescription ' maxValIdx1 = %d\n'],maxValIdx1);
470
471 if (config.receiverFilter)
472     receivedSignalBB1 = conv(raisedCosineSqrtFilter,receivedSignalBB1);
473     receivedSignalBB2 = conv(raisedCosineSqrtFilter,receivedSignalBB2);
474 end
475
476 PlotScatter( ...
477     receivedSignalBB1( ...
478         (maxValIdx1+rxFilterDelay) ...
479         + [leadInSequenceLength*symbolLength:complexSymbolsLength*symbolLength])/gain1, ...
480     receivedSignalBB2( ...
481         (maxValIdx1+rxFilterDelay) ...
482         + [leadInSequenceLength*symbolLength:complexSymbolsLength*symbolLength])/gain2, ...
483     symbolLength, multiRowFigureDetails,currentFilterTextDescription);
484
485 %-------------------------------------------------------------------------
486 % b. Re-sample at symbol locations (and at the fractional spacing rate)
487 %-------------------------------------------------------------------------
488 sampledSignal    = receivedSignalBB1(maxValIdx1 + rxFilterDelay ...
489                                      + [ ...
490                                          leadInSequenceLength*symbolLength ...
491                                          :symbolLength ...
492                                          :complexSymbolsLength*symbolLength])';
493
494 sampledSignalFS = receivedSignalBB1(maxValIdx1 + rxFilterDelay ...
495                                      + round([ ...
496                                          leadInSequenceLength*symbolLength ...
497                                          :symbolLength/adaptiveFIR.fractionalSteps ...
498                                          :complexSymbolsLength*symbolLength]))';
499
500 %-------------------------------------------------------------------------
501 % c. Pass the signal through the adaptive algorithms
502 %-------------------------------------------------------------------------
503 figureCount = figureCount + 1;
504 figure(figureCount);
505 set(figureCount,'Name',currentFilterTextDescription, ...
506                 'Position',[664   178   454   592]);
507 constellationValues = GetConstellationValues('PSK',0,txSignalInfo.PSK.modulationSettings);
508 adaptiveFilterData.tifpath =  ...
509     DuctExperimentRunAdaptiveTests(constellationValues,sampledSignal,sampledSignalFS, ...
510                                    trainingSequence,multiRowFigureDetails, ...
511                                    currentFilterTextDescription,adaptiveFIR);
512
513 %=========================================================================
514 % Plot the average signal error for each adaptive filter, and inverse filter type.
515 %=========================================================================
516 figure(multiRowFigureDetails.figureHandles.symbolErrorHistory);
517 adaptiveFilterFields = {'zf' 'mse' 'mseFS' 'rls' 'rlsFS'};
518 adaptiveFilterNames  = { ...
519     'Zero Forcing Algorithm (ZF)' ...
520     'Mean Square Error Algorithm (MSE)' ...
521     'Fractionally Spaced Mean Square Error Algorithm (MSE-FS)' ...
522     'Recursive Least Square (RLS)'...
523     'Fractionally Spaced Recursive Least Square Algorithm (RLS-FS)'};
524 adaptiveFilterSteps = { ...
525     adaptiveFIR.deltaSteps ...
526     adaptiveFIR.deltaSteps ...
```

```
527         adaptiveFIR.deltaStepsFS ...
528         adaptiveFIR.deltaStepsRLS ...
529         adaptiveFIR.deltaStepsRLS ...
530                         };
531     inverseFilterFields = {'mil2s' 'tr' 'tiffull' 'tifchan' 'tifpath'};
532     inverseFilterNames  = {'Stoj. 2–S' 'T.R.' 'T.I.F. (full)' 'T.I.F. (channel)' 'T.I.F. (path)'};
533     lineStyles = {'–' ':' '–.' '—' '–'};
534     grayColors = [ 0.7 0 0 0 0];
535     for ii = 1:length(adaptiveFilterFields)
536         subplot(length(adaptiveFilterFields),1,ii);
537         title(adaptiveFilterNames(ii));
538         hold on;
539         for jj = 1:length(inverseFilterFields)
540             plotValues(:,jj,ii) = getfield(...
541                 adaptiveFilterData,inverseFilterFields{jj},adaptiveFilterFields{ii},...
542                 'averageSignalError');
543             ph = plot(plotValues(:,jj,ii));
544             set(ph,'Marker','.','LineStyle',lineStyles{jj},'Color',grayColors(jj).*[1 1 1]);
545             set(gca,'XTick',1:6,'XTickLabel',adaptiveFilterSteps{ii})
546         end
547         hold off;
548     end
549     maxV = max(plotValues(:));
550     minV = min(plotValues(:));
551     upperLim = ceil(maxV*10^(-floor(log10(maxV)))) * 10^(floor(log10(maxV)));
552     lowerLim = floor(minV*10^(-floor(log10(minV)))) * 10^(floor(log10(minV)));
553     set(get(gcf,'children'),'YLim',[lowerLim upperLim],'YScale','log');
554     h = legend(inverseFilterNames,'Location','Best','Orientation','horizontal');
555     set(h,'Position',[0.05 0.03 0.9 0.035]);
```

A.2.4 Helper Scripts

A.2.4.1 DuctExperimentRunAdaptiveTests.m

```
1   function outData = DuctExperimentRunAdaptiveTests( ...
2       constellationValues,sampledSignal,sampledSignalFS,trainingSequence,multiRowFigureDetails, ...
3       currentFilterTextDescription,adaptiveFIR)
4   % USAGE: outData = DuctExperimentRunAdaptiveTests( ...
5   %    constellationValues,sampledSignal,sampledSignalFS,trainingSequence,multiRowFigureDetails, ...
6   %    currentFilterTextDescription,adaptiveFIR)
7   %
8   % This function is used as part of an experiment conducted in a duct that forms part of the thesis
9   % for Pierre Dumuid . This function runs the adaptive algorithms, 'Zero Forcing' (ZF), 'Mean Square
10  % Error' (MSE), and 'Recursive Least Square' (RLS) on the signals, sampledSignal, and
11  % sampledSignalFS. The signal, sampledSignal, is the sampled baseband complex signal that has been
12  % phase synchronised and sampled at the sampling rate, whilst the signal, sampledSignalFS, is the
13  % signal that has been phase synchronised and sampling at fractionalSteps times the sampling rate,
14  % and is used when performing fractional spacing equalisation. Fractional spacing equalisation is
15  % employed on for the RLS algorithm, and one of the instances of the RLS algorithm. The length of
16  % the feedforward, and feedback filters are given by feedbackFilterLength, and filterLength
17  % respectively. If the variable, trainingSequence contains a non–empty vector, then this vector
18  % provides a training sequence to initialise the filter taps.
19  %
20
21  colorGrey = 0.5*[1 1 1];
22  color2    = [1 0.3 0.7];
23
24  deltaSteps          = adaptiveFIR.deltaSteps;
25  deltaStepsFS        = adaptiveFIR.deltaStepsFS;
26  deltaStepsRLS       = adaptiveFIR.deltaStepsRLS;
27  fractionalSteps     = adaptiveFIR.fractionalSteps;
28  filterLength        = adaptiveFIR.filterLength;
```

```
29 feedbackFilterLength = adaptiveFIR.feedbackFilterLength;
30 filterPeakIndex       = adaptiveFIR.filterPeakIndex;
31
32 %=========================================================================
33 % Run non-recursive Detector with no stepsize (essentially straight detector)
34 %=========================================================================
35 zfa0.filterTaps = [1 zeros(1,filterLength)];
36 [zfa0.signalError zfa0.detectedSymbols zfa0.filterTaps zfa0.filterTapHistory zfa0.filteredSignal] ...
37     = DetectorAdaptiveZF(constellationValues,sampledSignal,0,zfa0.filterTaps,trainingSequence);
38
39 %=========================================================================
40 % Run Adaptive filters several times using different delta steps.
41 %=========================================================================
42 % Define the various parameter arrays
43 zfa.filterTapLongHistory     = [];
44 msea.filterTapLongHistory    = [];
45 rlsa.filterTapLongHistory    = [];
46 mseaFS.filterTapLongHistory = [];
47 rlsaFS.filterTapLongHistory = [];
48
49 zfa.signalErrorHistory       = [];
50 msea.signalErrorHistory      = [];
51 rlsa.signalErrorHistory      = [];
52 mseaFS.signalErrorHistory    = [];
53 rlsaFS.signalErrorHistory    = [];
54
55 zfa.filterTaps     = [1 zeros(1,filterLength)];
56 msea.filterTaps    = [zeros(1,filterPeakIndex-1) 1 zeros(1,filterLength-filterPeakIndex)];
57 rlsa.filterTaps    = [zeros(1,filterPeakIndex-1) 1 zeros(1,filterLength-filterPeakIndex)];
58 mseaFS.filterTaps = ...
59     [zeros(1,(filterPeakIndex-1)*fractionalSteps) ...
60      1 0 ...
61      zeros(1,(filterLength-filterPeakIndex)*fractionalSteps)];
62 rlsaFS.filterTaps = ...
63     [zeros(1,(filterPeakIndex-1)*fractionalSteps) ...
64      1 0 ...
65      zeros(1,(filterLength-filterPeakIndex)*fractionalSteps)];
66
67 msea.feedbackFilterTaps    = zeros(1,feedbackFilterLength);
68 rlsa.feedbackFilterTaps    = zeros(1,feedbackFilterLength);
69 mseaFS.feedbackFilterTaps = zeros(1,feedbackFilterLength);
70 rlsaFS.feedbackFilterTaps = zeros(1,feedbackFilterLength);
71
72 % Increment through the delta steps
73 for ii = 1:length(deltaSteps)
74     delta     = deltaSteps(ii);
75     deltaFS   = deltaStepsFS(ii);
76     deltaRLS = deltaStepsRLS(ii);
77
78     % Perform the adaptive filtering
79     [zfa.signalError zfa.detectedSymbols zfa.filterTaps zfa.filterTapHistory zfa.filteredSignal] ...
80         = DetectorAdaptiveZF(constellationValues,sampledSignal,delta,zfa.filterTaps,trainingSequence);
81
82     [msea.signalError msea.detectedSymbols msea.filterTaps msea.filterTapHistory ...
83      msea.filteredSignal msea.feedbackFilterTaps] = ...
84         DetectorAdaptiveMSE(constellationValues,sampledSignal,delta,msea.filterTaps, ...
85                             trainingSequence,1,msea.feedbackFilterTaps,filterPeakIndex);
86
87     [rlsa.signalError rlsa.detectedSymbols rlsa.filterTaps rlsa.filterTapHistory ...
88      rlsa.filteredSignal rlsa.feedbackFilterTaps] = ...
89         DetectorAdaptiveRLS(constellationValues,sampledSignal,deltaRLS,rlsa.filterTaps,...
90                             trainingSequence,1,rlsa.feedbackFilterTaps,filterPeakIndex);
91
92     [mseaFS.signalError mseaFS.detectedSymbols mseaFS.filterTaps mseaFS.filterTapHistory ...
93      mseaFS.filteredSignal mseaFS.feedbackFilterTaps] = ...
```

```
94          DetectorAdaptiveMSE(constellationValues,sampledSignalFS,deltaFS,mseaFS.filterTaps, ...
95                              trainingSequence,fractionalSteps,mseaFS.feedbackFilterTaps, ...
96                              filterPeakIndex);
97
98      [rlsaFS.signalError rlsaFS.detectedSymbols rlsaFS.filterTaps rlsaFS.filterTapHistory ...
99       rlsaFS.filteredSignal rlsaFS.feedbackFilterTaps] = ...
100         DetectorAdaptiveRLS(constellationValues,sampledSignalFS,deltaRLS,rlsaFS.filterTaps, ...
101                             trainingSequence,fractionalSteps,rlsaFS.feedbackFilterTaps, ...
102                             filterPeakIndex);
103
104     fprintf('%.4f ZFA: %.4f MSEA: %.4f RLSA: %.4f MSEA-FS %.4f RLSA-FS: %.4f\n',delta, ...
105             mean(abs(zfa.signalError)),mean(abs(msea.signalError)),mean(abs(rlsa.signalError)), ...
106             mean(abs(mseaFS.signalError)),mean(abs(rlsaFS.signalError)));
107
108     % Keep a history of the results
109     zfa.signalErrorHistory          = [   zfa.signalErrorHistory     zfa.signalError];
110     msea.signalErrorHistory         = [  msea.signalErrorHistory    msea.signalError];
111     rlsa.signalErrorHistory         = [  rlsa.signalErrorHistory    rlsa.signalError];
112     mseaFS.signalErrorHistory       = [mseaFS.signalErrorHistory mseaFS.signalError];
113     rlsaFS.signalErrorHistory       = [rlsaFS.signalErrorHistory rlsaFS.signalError];
114     zfa.averageSignalError(ii)      = mean(abs(    zfa.signalError));
115     msea.averageSignalError(ii)     = mean(abs(   msea.signalError));
116     rlsa.averageSignalError(ii)     = mean(abs(   rlsa.signalError));
117     mseaFS.averageSignalError(ii) = mean(abs(mseaFS.signalError));
118     rlsaFS.averageSignalError(ii) = mean(abs(rlsaFS.signalError));
119     zfa.filterTapLongHistory        = [   zfa.filterTapLongHistory     zfa.filterTapHistory];
120     msea.filterTapLongHistory       = [  msea.filterTapLongHistory    msea.filterTapHistory];
121     rlsa.filterTapLongHistory       = [  rlsa.filterTapLongHistory    rlsa.filterTapHistory];
122     mseaFS.filterTapLongHistory     = [mseaFS.filterTapLongHistory mseaFS.filterTapHistory];
123     rlsaFS.filterTapLongHistory     = [rlsaFS.filterTapLongHistory rlsaFS.filterTapHistory];
124 end
125 fprintf('\n');
126
127 %==========================================================================
128 % Plot the results
129 %==========================================================================
130 currentFigureHandle = gcf;
131
132 %--------------------------------------------------------------------------
133 % Plot the scatter plots for each adaptive filter
134 %--------------------------------------------------------------------------
135 figure(multiRowFigureDetails.figureHandles.adaptiveScatterPlots);
136 totalRows  = multiRowFigureDetails.totalRows;
137 currentRow = multiRowFigureDetails.currentRow;
138 subplot(totalRows,1,currentRow);
139 plotSeperator = 3;
140 hold off;
141 lh = plot(zfa0.filteredSignal(1:end-1)    + 1*plotSeperator,'k.');
142 hold on;
143 plot(zfa.filteredSignal(1:end-1)     + 2*plotSeperator,'k.');
144 plot(msea.filteredSignal(1:end-1)    + 3*plotSeperator,'k.');
145 plot(mseaFS.filteredSignal(1:end-1) + 4*plotSeperator,'k.');
146 plot(rlsa.filteredSignal(1:end-1)    + 5*plotSeperator,'k.');
147 plot(rlsaFS.filteredSignal(1:end-1) + 6*plotSeperator,'k.');
148 set(get(gca,'children'),'MarkerSize',2);
149 set(gca,'XTick',[1:6]*plotSeperator, ...
150         'XTickLabel',{'none','ZF','MSE','MSE-FS','RLS','RLS-FS'}, ...
151         'XLim',[plotSeperator - 2 6*plotSeperator+2]);
152 hold off
153 ylabel(currentFilterTextDescription);
154
155 %--------------------------------------------------------------------------
156 % Plot filter tap history
157 %--------------------------------------------------------------------------
158 figure(currentFigureHandle)
```

```
subplot(5,1,1);
imagesc(20*log10(abs(zfa.filterTapLongHistory)))
title('ZF');
subplot(5,1,2);
imagesc(20*log10(abs(msea.filterTapLongHistory)))
title('MSE');
subplot(5,1,3);
imagesc(20*log10(abs(mseaFS.filterTapLongHistory)))
title('MSE-FS');
subplot(5,1,4);
imagesc(20*log10(abs(rlsa.filterTapLongHistory)))
title('RLS');
subplot(5,1,5);
imagesc(20*log10(abs(  rlsaFS.filterTapLongHistory)))
title('RLS-FS');
xlabel('samples');

outData.zf.averageSignalError     = zfa.averageSignalError;
outData.mse.averageSignalError    = msea.averageSignalError;
outData.mseFS.averageSignalError = mseaFS.averageSignalError;
outData.rls.averageSignalError    = rlsa.averageSignalError;
outData.rlsFS.averageSignalError = rlsaFS.averageSignalError;

outData.zf0.filteredSignal    = zfa0.filteredSignal(1:end-1);
outData.zf.filteredSignal     = zfa.filteredSignal(1:end-1);
outData.mse.filteredSignal    = msea.filteredSignal(1:end-1);
outData.mseFS.filteredSignal = mseaFS.filteredSignal(1:end-1);
outData.rls.filteredSignal    = rlsa.filteredSignal(1:end-1);
outData.rlsFS.filteredSignal = rlsaFS.filteredSignal(1:end-1);
```

A.2.4.2 GetDS1104VariableDescriptions.m

```
function dSpaceMapVar = GetDS1104VariableDescriptions()
% USAGE: dSpaceMapVar = GetDS1104VariableDescriptions()
%
%  This function generates a structure containing the map varibles used by the dSPACE function,
%  mlib() to retrieve the variables from the dSpace DS1104 board. These variable descriptions are
%  specific to the program, DuctExperimentDSPACEProgram.c that must be loaded onto the DS1104
%  device.
%

%=========================================================================================
% Program execution variables
%=========================================================================================
dSpaceMapVar.program_status       = mlib('GetMapVar','program_status'       ,'type','UInt16');
dSpaceMapVar.program_control      = mlib('GetMapVar','program_control'      ,'type','Int16');

%=========================================================================================
% General variables
%=========================================================================================
dSpaceMapVar.data_block           = mlib('GetMapVar','data_block'           ,'type','Float64');
dSpaceMapVar.sample_period        = mlib('GetMapVar','sample_period');

%=========================================================================================
% Routine variables
%=========================================================================================
dSpaceMapVar.play_sample_length   = mlib('GetMapVar','play_sample_length'   ,'type','UInt32');
dSpaceMapVar.record_sample_length = mlib('GetMapVar','record_sample_length','type','UInt32');
dSpaceMapVar.play_channels        = mlib('GetMapVar','play_channels'        ,'type','UInt16' ,'length',9);
dSpaceMapVar.record_channels      = mlib('GetMapVar','record_channels'      ,'type','UInt16' ,'length',9);
dSpaceMapVar.play_channel_gains   = mlib('GetMapVar','play_channel_gains'   ,'type','Float64','length',8);
dSpaceMapVar.record_channel_gains = mlib('GetMapVar','record_channel_gains','type','Float64','length',8);

```

```
%=========================================================================
% Variable to write bits to the i/o connector
%=========================================================================
dSpaceMapVar.io_bits_write          = mlib('GetMapVar','io_bits_write'          ,'type','UInt16' ,'length',1);
```

A.2.4.3 GetFakeIRFs.m

```
function outIRFs = GetFakeIRFs(playChannelsCount,recordChannelsCount,filterLength)
% USAGE: outIRFs = GetFakeIRFs(playChannelsCount,recordChannelsCount,filterLength)
%
% Make a set of fake impulse using random values enveloped in decaying exponent envelopes.
%

noEnvelopes = 6;
for ii = 1:recordChannelsCount
    for jj =  1:playChannelsCount
        outIRFs(:,ii,jj) = zeros(filterLength,1);
        for envelopeAdditionIndex = 1:noEnvelopes
            outIRFs(:,ii,jj) = outIRFs(:,ii,jj) ...
                + (rand(filterLength,1)-0.5).*exp(-(3+rand*4)*[1:filterLength]./filterLength).';
        end
    end
end
```

A.2.4.4 GetIRFsDS1104.m

```
function [outIRFs coherence noise] = ...
        GetIRFsDS1104(freqSampling,playChannels,recordChannels,channelSettings)
% USAGE: [outIRFs coherence noise] = ...
%             GetIRFsDS1104(freqSampling,playChannels,recordChannels,channelSettings)
%
% Description:
%   This script is used to characterise the impulse response between a set of outputs and inputs
%   connected to the dSPACE board.
%
% Before invoking this M-file the real-time processor application DuctExperimentDSPACEProgram.obj
% (or equivalent) must be loaded and running on the dSPACE board.

if (nargin < 2)
    playChannels    = 1:6;
    recordChannels = 1:6;
end

%=========================================================================
% Set parameters on dSPACE board
%=========================================================================
mlib('SelectBoard','DS1104')

deviceSettings.freqSampling = freqSampling;
deviceSettings.playSampleLength = 43000;

dSpaceMapVar = GetDS1104VariableDescriptions;

% Set the output bits so that the switch box switches to the current arrays.
outBits = 1*(channelSettings.recordArrayId-1) + 2*(channelSettings.playArrayId-1);
mlib('write',dSpaceMapVar.io_bits_write          ,'data',outBits);

% Load the current system gains
if isfield(channelSettings,'playGains')
    playGains = channelSettings.playGains;
else
```

```
36      b = load('spkgains');
37      currentPlayGains = b.spkgains.current;
38      if channelSettings.playArrayId == 1
39          playGains = currentPlayGains.tx1;
40      else
41          playGains = currentPlayGains.tx2;
42      end
43  end
44  if isfield(channelSettings,'recordGains')
45      playGains = channelSettings.recordGains;
46  else
47      a = load('micgains');
48      currentRecordGains = a.micgains.current;
49      if channelSettings.recordArrayId == 1
50          recordGains = currentRecordGains.rx1;
51      else
52          recordGains = currentRecordGains.rx2;
53      end
54  end
55
56  % Set the record and playback gains on the DS1104 device.
57  playChannelGains    = playGains(playChannels);
58  playChannelGains    = [playChannelGains zeros(1,8-length(playChannelGains))];
59  recordChannelGains = recordGains(recordChannels);
60  recordChannelGains = [recordChannelGains zeros(1,8-length(recordChannelGains))];
61  mlib('write',dSpaceMapVar.record_channel_gains,'data',recordChannelGains);
62  mlib('write',dSpaceMapVar.play_channel_gains   ,'data',playChannelGains);
63
64
65  % Set the settings for the RunDS1104ChkSpk routine.
66  deviceSettings.playChannels    = playChannels;
67  deviceSettings.recordChannels = recordChannels;
68  deviceSettings.playChannelsCount    = length(deviceSettings.playChannels);
69  deviceSettings.recordChannelsCount = length(deviceSettings.recordChannels);
70
71  %=========================================================================
72  % Create a signal to characterise the signal with.
73  %=========================================================================
74  numberZeroForNoiseFloor       = 1024*6; % Zero at the beginning of the signal used to measure the
75                                          % noise in the system.
76  numberZerosBeforeNextChannel = 1500;    % Zeros at end of signal to avoid influencing the
77                                          % measurements for the next speaker.
78
79  playSampleNonZeroLength = deviceSettings.playSampleLength - numberZerosBeforeNextChannel;
80
81  % Generate the random signal used to characterise the system.
82  playSample = rand(1,playSampleNonZeroLength);
83
84  % Normalise and zero pad to make sure it is sampleLength samples long.
85  playSample = (playSample - mean(playSample));
86  playSample = playSample / sqrt(max(playSample.^2));
87  playSample = [playSample(1:playSampleNonZeroLength)'; ...
88                zeros(deviceSettings.playSampleLength - playSampleNonZeroLength,1)];
89  playSample = playSample(1:deviceSettings.playSampleLength);
90
91  % Add the zeros to the beginning of the sample
92  playSample = [zeros(numberZeroForNoiseFloor,1); ...
93                playSample(numberZeroForNoiseFloor+1:end-4400); ...
94                zeros(4400,1)];
95
96  %=========================================================================
97  % Play the characterisation signal
98  %=========================================================================
99  % Pause a little bit to allow the input switch to settle if it was switched
100 pause(.25);
```

```
101
102 disp('Now Performing measurement to determine system outIRFs');
103 recordData = RunDS1104Chkspk(deviceSettings, playSample);
104
105 %=========================================================================
106 % Modify the received signal
107 %=========================================================================
108 % reshape the recordData (Dimensions: 1=sample Measurements, 2=RX.No.+Time, 3=TX.No.)
109 recordData = reshape(recordData(deviceSettings.playSampleLength + 1:end), ...
110                      deviceSettings.playSampleLength, ...
111                      deviceSettings.recordChannelsCount + 1, ...
112                      deviceSettings.playChannelsCount);
113
114 noise = recordData(1:numberZeroForNoiseFloor, ...
115                    1:deviceSettings.recordChannelsCount, ...
116                    1:deviceSettings.playChannelsCount);
117
118 % Ensure that the inputs haven't maxed out...
119 maxValues = squeeze(max(abs(recordData),[],1))';
120 maxValues = maxValues(:,1:deviceSettings.recordChannelsCount);
121 maxValue = max(max(maxValues*diag(recordGains(recordChannels).^-1)));
122 fprintf('max value was %.3f\n',maxValue);
123 if maxValue > 0.9
124     disp('WARNING: Level overload occurred')
125 end
126
127 %=========================================================================
128 % Determine the impulse responses using the FFT method.
129 %=========================================================================
130 disp('Now Determining IRF''s via FFT method')
131 fprintf('\noutIRFs:');
132
133 % Create a matrix of the transmitted recordData (with the dely removed...)
134 transmitSignal = playSample(numberZeroForNoiseFloor+1:end);
135
136 % ==> FFT Method Variables
137 % Using the FFT method:
138
139 % Number of points in the FFT
140 fftSettings.n = 2048*4;
141
142 % Number of points to calculate FFT over
143 fftSettings.m = (size(playSample) ...
144                  - numberZeroForNoiseFloor ...
145                  - numberZerosBeforeNextChannel ...
146                  - 1000);
147
148 % Overlap to use in the FFT
149 fftSettings.overlap = floor(fftSettings.n*0.25);
150
151 tic
152 for ii = 1:deviceSettings.recordChannelsCount
153     for jj = 1:deviceSettings.playChannelsCount
154         fprintf('.');
155         % Isolate the recorded data for reciever ii,jj
156         receivedSignal = recordData(numberZeroForNoiseFloor+1:end,ii,jj);
157
158         % Use the MATLAB command, spectrum to perform the FFT calculations
159         P = spectrum(transmitSignal([1:fftSettings.m]), ...
160                      receivedSignal(1:fftSettings.m), ...
161                      fftSettings.n, ...
162                      fftSettings.overlap);
163
164         % The spectrum command returns the frequency response for positive frequencies.
165         % Given the outIRFs must be real, the negative frequecies can be made from the conjugate
```

```
166             fullFrequencyResponse(:,ii,jj) = [P(:,4); conj(P([end-1:-1:2],4))];
167             outIRFs(:,ii,jj) = real(ifft(fullFrequencyResponse(:,ii,jj)));
168 
169             coherence(:,ii,jj) = P([1:end end-1:-1:2],5);
170         end
171         fprintf('%d',ii);
172     end
173     fprintf('  (%f)\n',toc);
```

A.2.4.5 PlotScatter.m

```
1   function PlotScatter(complexSignal1,complexSignal2,symbolLength,multiRowFigureDetails,...
2       currentFilterTextDescription)
3   % USAGE: PlotScatter(complexSignal1,complexSignal2,symbolLength,multiRowFigureDetails,...
4   %   currentFilterTextDescription)
5   %
6   % This function creates two scatter plots in one figure. The plots contain x's at for each sample
7   % seperated by symbolLength, and a grey signal elsewhere. It is used to plot scatter plots of two
8   % complex signals (complexSignal1 and complexSignal2) that have sampling points seperated by
9   % symbolLength.
10  %
11  totalRows  = multiRowFigureDetails.totalRows;
12  currentRow = multiRowFigureDetails.currentRow;
13  figure(multiRowFigureDetails.figureHandles.scatterPlots);
14  
15  subplot(totalRows,2,(currentRow-1) * 2 + 1);
16  %lineHandle = plot(real(complexSignal1),imag(complexSignal1),'-');
17  %set(lineHandle,'Color',[1 1 1]*.8)
18  %hold on
19  lh = plot(real(complexSignal1(1:symbolLength:end)),imag(complexSignal1(1:symbolLength:end)),'k.');
20  set(lh,'MarkerSize',2);
21  %hold off
22  if currentRow == 1;
23      title('Target receiver');
24  end
25  ylabel(currentFilterTextDescription);
26  axis([-1 1 -1 1]*max(abs(axis)));
27  axesHandle = axis;
28  
29  subplot(totalRows,2,(currentRow-1) * 2 + 2);
30  lineHandle = plot(real(complexSignal2),imag(complexSignal2),'-');
31  set(lineHandle,'Color',[1 1 1]*.8)
32  lh = plot(real(complexSignal2(1:symbolLength:end)),imag(complexSignal2(1:symbolLength:end)),'k.');
33  set(lh,'MarkerSize',2);
34  if currentRow == 1;
35      title('Non-target receiver');
36  end
37  ylabel(' ');
38  axis(axesHandle);
```

A.2.4.6 RunDS1104Chkspk.m

```
1   function recordData = RunDS1104Chkspk(deviceSettings,playSample)
2   % USAGE: recordData = RunDS1104Chkspk(deviceSettings,playSample)
3   %
4   % This function is designed to play the same vector on each output channel, one after the other and
5   % record the response on all the inputs. (i.e. stepped single-input, multi-output)
6   %
7   % The variables are:
8   %   deviceSettings - a structure with fields:
9   %                    .playChannels       - matrix of the channels to play
```

```
%                   .recordChannels      - matrix of the channels to record
%                   .playChannelsCount   - number of channels to play
%                   .recordChannelsCount - number of channels to record
%                   .freqSampling        - sampling frequency
%     playSample - The signal to play
%     recordData - The recorded signal:
%       0                    -  1*playSampleLength-1 : Sample to play
%       1*playSampleLength   -  2*playSampleLength-1 : Channel 1 response for playing on channel 1
%       2*playSampleLength   -  3*playSampleLength-1 : Channel 2 response for playing on channel 1
%       ...
%       6*playSampleLength   -  7*playSampleLength-1 : Channel 6 response for playing on channel 1
%       7*playSampleLength   -  8*playSampleLength-1 : Time samples for playing on channel 1
%       9*playSampleLength   - 10*playSampleLength-1 : Channel 1 response for playing on channel 2
%       10*playSampleLength  - 11*playSampleLength-1 : Channel 2 response for playing on channel 2
%       ...
%       14*playSampleLength  - 15*playSampleLength-1 : Channel 6 response for playing on channel 2
%       15*playSampleLength  - 16*playSampleLength-1 : Time samples for playing on channel 2
%       ...
%       ...
%       etc.
%
%      NOTE: The above is an example for recording on 6 channels, if recording on a different number
%       of channels, the offsets for the data is equivalently adjusted.
%

%==========================================================================
% Set parameters on dSPACE board
%==========================================================================
mlib('SelectBoard','DS1104');

% Get the variable descriptions
dSpaceMapVar = GetDS1104VariableDescriptions();

% Set which channels to play and record on
playChannels   = [deviceSettings.playChannels(:); zeros(8,1)]';
recordChannels = [deviceSettings.recordChannels(:);  zeros(8,1)]';
mlib('write',dSpaceMapVar.play_channels, ...
      'data',[deviceSettings.playChannelsCount playChannels(1:8)]);
mlib('write',dSpaceMapVar.record_channels, ...
      'data',[deviceSettings.recordChannelsCount  recordChannels(1:8)]);

% Set the sampling period
mlib('Write',dSpaceMapVar.sample_period,'Data',1/deviceSettings.freqSampling);

% Set the play length
playSampleLength = length(playSample);
mlib('Write',dSpaceMapVar.play_sample_length,'Data',playSampleLength);

% Write the playSample data to the datablock
dSpaceMapVar.data_block.length = playSampleLength;
mlib('Write', dSpaceMapVar.data_block, 'Data', playSample);

% Set board to state 3:Play_Stepped_SIMO
mlib('Write', dSpaceMapVar.program_control,'Data',3)
mlib('Write', dSpaceMapVar.program_status ,'Data',0)

%==========================================================================
% Play the signal on each speaker
%==========================================================================
mlib('Intrpt');

% Wait 60% of the time that the program should take.
pause(0.6 * deviceSettings.playChannelsCount * playSampleLength / deviceSettings.freqSampling);

while (mlib('Read',dSpaceMapVar.program_status) ~= 2)
```

```
75      % Waiting for the process to finish
76  end
77
78  %=========================================================================
79  % Retrieve the recorded signals
80  %=========================================================================
81  % Read in the sample that was played
82  tempDSpaceMapVar          = dSpaceMapVar.data_block;
83  tempDSpaceMapVar.length = playSampleLength;
84  recordData                = mlib('Read',tempDSpaceMapVar);
85
86  % Increment the address to the location of the first recording.
87  tempDSpaceMapVar.addr   = tempDSpaceMapVar.addr + playSampleLength * 8;
88  tempDSpaceMapVar.length = playSampleLength * (deviceSettings.recordChannelsCount + 1);
89
90  % Read the responsed for each played channel
91  for ii = 1:deviceSettings.playChannelsCount
92      recordData             = [recordData mlib('Read',tempDSpaceMapVar)];
93      tempDSpaceMapVar.addr = tempDSpaceMapVar.addr + tempDSpaceMapVar.length * 8;
94  end
```

A.2.4.7 RunDS1104MIMO.m

```
1   function [recordData recordSamplingTimes] = RunDS1104MIMO(deviceSettings,playSample)
2   % USAGE: [recordData recordSamplingTimes] = RunDS1104MIMO(deviceSettings,playSample)
3   %
4   % This function is designed to simultaneously play a set of signals on the dSpace outputs and record
5   % the response on the inputs.
6   %
7   % The variables are:
8   %     deviceSettings - a structure with fields:
9   %                      .playChannels       - matrix of the channels to play
10  %                      .recordChannels     - matrix of the channels to record
11  %                      .playChannelsCount  - number of channels to play
12  %                      .recordChannelsCount - number of channels to record
13  %                      .freqSampling       - sampling frequency
14  %                      .playSampleLength   - the length of the signal to play
15  %                      .recordSampleLength - the length of the signals to record
16  %     playSample - The signal to play
17  %     recordData - The recorded signal:
18  %                          0                      - 1*playSampleLength-1    : Output 1 Sample
19  %                          1*playSampleLength     - 2*playSampleLength-1    : Output 2 Sample
20  %                            ...
21  %                        (N-1)*playSampleLength   - N*playSampleLength-1    : Output N Sample
22  %
23  %                      After recordOffset:
24  %                          0                      - 1*recordSampleLength-1 : Input 1 Signal
25  %                          1*recordSampleLength - 2*recordSampleLength-1 : Input 2 Signal
26  %                            ...
27  %                        (M-1)*recordSampleLength - M*recordSampleLength-1 : Input 3 Signal
28  %
29  %                      After timeOffset:
30  %                          0                      - maxSampleLength         : Sampling times
31  %
32  %                      where
33  %                        recordOffset    = N*playSampleLength
34  %                        timeOffset      = N*playSampleLength + M*recordSampleLength
35  %                        maxSampleLength = max(recordSampleLength, playSampleLength)
36  %     recordsampleTimes - The sampling times
37
38  % Reshape data-block if incorrect
39  [playSampleSize1 playSampleSize2] = size(playSample);
40  if (playSampleSize2 == 2)
```

```
41      playSample2 = playSample.';
42  else
43      playSample2 = playSample;
44  end
45
46  %=========================================================================
47  % Set parameters on dSPACE board
48  %=========================================================================
49  mlib('SelectBoard','DS1104');
50
51  % Get the variable descriptions
52  dSpaceMapVar = GetDS1104VariableDescriptions();
53
54  % Set which channels to play and record on
55  playChannels   = [deviceSettings.playChannels(:); zeros(8,1)]';
56  recordChannels = [deviceSettings.recordChannels(:);  zeros(8,1)]';
57  mlib('write',dSpaceMapVar.play_channels, ...
58        'data',[deviceSettings.playChannelsCount playChannels(1:8)]);
59  mlib('write',dSpaceMapVar.record_channels, ...
60        'data',[deviceSettings.recordChannelsCount  recordChannels(1:8)]);
61
62  % Set the sampling period
63  mlib('Write',dSpaceMapVar.sample_period,'Data',1/deviceSettings.freqSampling);
64
65  % Set the record and play lengths
66  mlib('Write',dSpaceMapVar.play_sample_length,'Data',deviceSettings.playSampleLength);
67  mlib('Write',dSpaceMapVar.record_sample_length,'Data',deviceSettings.recordSampleLength);
68
69  % Write the playSample data to the datablock
70  localDSpaceMapVar.play_data_block = dSpaceMapVar.data_block;
71  localDSpaceMapVar.play_data_block.length = ...
72      deviceSettings.playSampleLength * deviceSettings.playChannelsCount;
73  localDSpaceMapVar.play_data_block.offset = 0;
74  mlib('Write', localDSpaceMapVar.play_data_block, 'Data', playSample2(:));
75
76  % Set board to state 4:Play_MIMO
77  mlib('Write', dSpaceMapVar.program_control,'Data',4)
78  mlib('Write', dSpaceMapVar.program_status ,'Data',0)
79
80  %=========================================================================
81  % Play the signal on the speakers
82  %=========================================================================
83  mlib('Intrpt');
84
85  % Wait 60% of the time that the program should take.
86  pause(0.6 ...
87        * max([deviceSettings.recordSampleLength deviceSettings.playSampleLength]) ...
88        / deviceSettings.freqSampling);
89
90  while (mlib('Read',dSpaceMapVar.program_status) ~= 2)
91      % Waiting for the process to finish
92  end
93
94  %=========================================================================
95  % Retrieve the recorded signals
96  %=========================================================================
97  localDSpaceMapVar.record_data_block = dSpaceMapVar.data_block;
98  localDSpaceMapVar.record_data_block.offset = ...
99      deviceSettings.playChannelsCount * deviceSettings.playSampleLength;
100 localDSpaceMapVar.record_data_block.length = deviceSettings.recordSampleLength;
101
102 for ii = 1:deviceSettings.recordChannelsCount
103     recordData(:,ii) = mlib('read',localDSpaceMapVar.record_data_block).';
104     localDSpaceMapVar.record_data_block.offset = localDSpaceMapVar.record_data_block.offset ...
105         + deviceSettings.recordSampleLength;
```

```
106 end
107
108 % Read the Recording Sampling Times
109 if nargout > 1
110     localDSpaceMapVar.time_data_block = dSpaceMapVar.data_block;
111     localDSpaceMapVar.time_data_block.length = ...
112         max([deviceSettings.recordSampleLength deviceSettings.playSampleLength]);
113     localDSpaceMapVar.time_data_block.offset = ...
114         deviceSettings.playChannelsCount * deviceSettings.playSampleLength ...
115         + deviceSettings.recordChannelsCount * deviceSettings.recordSampleLength;
116     recordSamplingTimes = mlib('read',localDSpaceMapVar.time_data_block);
117 end
```

A.3 Code used for the simulation

A.3.1 The Condor submitter scripts

A.3.1.1 DuctSimulationManager.m

```
1  % DuctSimulationManager.m
2  %
3  % USAGE:
4  %         DuctSimulationManager
5  %
6  % This script is used to manage the simulation the results that would be obtained for the
7  % communication systems performed in duct that forms part of the thesis for Pierre Dumuid.
8  %
9  % Prior to executing the script, the following variables should be defined in the MATLAB:
10 %
11 %         simulationConfig.arrangementIndex
12 %             - The index of the arrangement being simulated
13 %         simulationConfig.symbolLengthIndex
14 %             - The index of the symbol length being simulated
15 %
16 %         managerConfig.refetchResults
17 %             - If this script should execute the script to submit the jobs and fetch the results.
18 %         managerConfig.plotGraphs
19 %
20 %         submitJobsAndFetchResultsConfig.resubmitCondorJobs
21 %             - Resubmit a condor job if results are not present
22 %         submitJobsAndFetchResultsConfig.usePreviousFetchedResult
23 %             - Use previously fetched results if data exists from last fetch.
24 %
25
26 %==========================================================================
27 % Define variables
28 %==========================================================================
29
30 submitJobsAndFetchResultsConfig.fakeMissingResults        = 0;
31 submitJobsAndFetchResultsConfig.submitJobsInRandomOrder = 1;
32
33 defaults.plotResultsConfig.showMainResults    = 1;
34 defaults.plotResultsConfig.showTXPowerVSPeak = 0;
35 defaults.plotResultsConfig.showCondorTime     = 0;
36 defaults.plotResultsConfig.annotationPosition     = [0.105 0.71 1 0.02];
37 defaults.plotResultsConfig.colorbarXPosition      = [0.95 0.02];
38 defaults.plotResultsConfig.timeReversalYPosition = [0.06 0.02];
39 defaults.plotResultsConfig.subplotGridPosition    = [.058 0.065 .88 .825];
40 defaults.plotResultsConfig.subplotGridSpacing     = [0.005 0.01];
41
42 % Apply defaults plotResultConfigs if none are set
```

```
43 defaultFieldNames = fieldnames(defaults.plotResultsConfig);
44 for fii = 1:length(defaultFieldNames);
45     defaultFieldName = defaultFieldNames{fii};
46     if ~isfield(plotResultsConfig,defaultFieldName)
47         plotResultsConfig = setfield( ...
48             plotResultsConfig, ...
49             defaultFieldName, ...
50             getfield(defaults.plotResultsConfig,defaultFieldName) ...
51             );
52     end
53 end
54
55 %------------------------------------------------------------------------
56 % Static variables
57 %------------------------------------------------------------------------
58
59 % The ranges of the parameters that the system is tested over
60 staticConfig.ranges.frequencyCarrier     = [1.5e3:27.8320:20e3];
61 staticConfig.ranges.beta                 = [0 10.^[-20:0.5:0]];
62 staticConfig.ranges.symbolLength         = [25 10 5];
63 staticConfig.ranges.transmitterElements = { ...
64     [1 2 3 4 5 6], ...
65     [1 2 3], ...
66     [1 3] ...
67                     };
68
69 % Information concerning the inverse filters that are tested.
70 staticConfig.inverseFilters.names = { ...
71     'Time Reversal' ...
72     'Stojanovic''s Two-sided' ...
73     'Inverse Filter: Path' ...
74     'Inverse Filter: Channel' ...
75     'Inverse Filter: Full MIMO' ...
76                     };
77
78 staticConfig.inverseFilters.abbreviations = { ...
79     'T.R.' ...
80     'Stoj. 2sided' ...
81     'TIF: Path' ...
82     'TIF: Chan' ...
83     'TIF: MIMO' ...
84                     };
85 staticConfig.inverseFilters.textKeys = { ...
86     'tr' ...
87     'mil2s' ...
88     'if_path' ...
89     'if_chan' ...
90     'if_fmimo' ...
91                     };
92 staticConfig.inverseFilters.betaVaries = [ ...
93     0 ...
94     1 ...
95     1 ...
96     1 ...
97     1 ...
98                     ];
99
100 % Keys for the general results obtained from each simulation
101 staticConfig.results.generalMeasureVariableNames = {
102     'prerx_power1' ...
103     'prerx_power2' ...
104     'prerx_power' ...
105     'prerx_power1_t' ...
106     'prerx_power2_t' ...
107     'prerx_power_t' ...
```

```
108       'postrx_power'          ...
109       'postrx_power_t'        ...
110       'symbolerr'             ...
111       'txpeak'                ...
112       'txpower'               ...
113       'gain'                  ...
114       'idx'                   ...
115       'total_std'             ...
116       'total_std2'            ...
117       'crosstalk_prerx'       ...
118       'crosstalk_postrx'      ...
119       'crosstalk_postrx_rng' ...
120                     };
121
122   % Keys for the adaptive filters used in the simulations
123   staticConfig.results.adaptiveFilterKeys         = {'zfa' 'zfat' 'msea' 'mseat'};
124
125   % Keys for the properties measured from the adaptive filters
126   staticConfig.results.adaptiveFilterMeasureKeys = {'_ee' '_std' '_symbolerr'};
127
128   %-------------------------------------------------------------------------------------------
129   % Condor submit file settings
130   %-------------------------------------------------------------------------------------------
131   submitFileSettings.environmentPath           ='C:\\WINNT\\system32;C:\\windows\\system32;D:\\temp;';
132   submitFileSettings.notify                    = 0;
133   submitFileSettings.additionalTransferFiles = '';
134   submitFileSettings.executable                = '..\\..\\pierres4job_pf.bat';
135   submitJobsAndFetchResultsConfig.submitFileSettings = submitFileSettings;
136
137   %-------------------------------------------------------------------------------------------
138   % Condor job mat file settings
139   %-------------------------------------------------------------------------------------------
140   % The variable, runparams, is loaded by each the condor job to control the parameters that are
141   % tested.
142   runparams.matpath = '/home/pmdumuid/MyDocs/lyx/thesis/ExperimentCode/Experiment3/matFiles/';
143   runparams.runopts.do.tr        = 1;
144   runparams.runopts.do.mil2s     = 1;
145   runparams.runopts.do.if_path   = 1;
146   runparams.runopts.do.if_chan   = 1;
147   runparams.runopts.do.if_fmimo = 1;
148   runparams.runopts.isfd         = 0;
149
150   %-------------------------------------------------------------------------------------------
151   % Derived variable values
152   %-------------------------------------------------------------------------------------------
153
154   % The following runparams are determined from the staticConfig and simulationConfig values.
155   runparams.betarange      = staticConfig.ranges.beta;
156   runparams.symblen        = staticConfig.ranges.symbolLength(simulationConfig.symbolLengthIndex);
157   runparams.IRFchannelidx = staticConfig.ranges.transmitterElements{simulationConfig.arrangementIndex};
158
159   % Name of variable to hold all the results
160   alphabetLetters = {'c' 'd' 'e'};
161   mergedJobResultsVariableName = sprintf( ...
162       ['sim2a_results%02d' alphabetLetters{simulationConfig.symbolLengthIndex}], ...
163       simulationConfig.arrangementIndex);
164
165   % Name of the job (used for the directory name)
166   mergedJobName = sprintf( ...
167       ['sim2_pf%02d' alphabetLetters{simulationConfig.symbolLengthIndex}], ...
168       simulationConfig.arrangementIndex);
169
170   % Created the selectedInverseFilters variable.
171   selectedFilterIndexes = [];
172   if (runparams.runopts.do.tr)        selectedFilterIndexes = [selectedFilterIndexes 1]; end
```

```
if (runparams.runopts.do.mil2s)     selectedFilterIndexes = [selectedFilterIndexes 2]; end
if (runparams.runopts.do.if_path)   selectedFilterIndexes = [selectedFilterIndexes 3]; end
if (runparams.runopts.do.if_chan)   selectedFilterIndexes = [selectedFilterIndexes 4]; end
if (runparams.runopts.do.if_fmimo) selectedFilterIndexes = [selectedFilterIndexes 5]; end
selectedInverseFilters.textKeys    = staticConfig.inverseFilters.textKeys(selectedFilterIndexes);
selectedInverseFilters.names       = staticConfig.inverseFilters.names(selectedFilterIndexes);
selectedInverseFilters.betaVaries = staticConfig.inverseFilters.betaVaries(selectedFilterIndexes);
selectedInverseFilters.abbreviations = ...
    staticConfig.inverseFilters.abbreviations(selectedFilterIndexes);

%=========================================================================
% (Re-)Fetch the results
%=========================================================================
if (managerConfig.refetchResults)
    tst = ~exist(mergedJobResultsVariableName);
    % Initialise the variable, resultsFetched if the variable,mergedJobResultsVariableName, isn't
    % defined yet.
    if (tst)
        eval([mergedJobResultsVariableName '.resultsFetched = ' ...
              'zeros(1,length(staticConfig.ranges.frequencyCarrier));']);
    end
    eval(['[' mergedJobResultsVariableName ' resultInfo] = ' ...
          'DuctSimulationSubmitJobsAndFetchResults(' mergedJobResultsVariableName ',' ...
          'submitJobsAndFetchResultsConfig,simulationConfig,runparams,mergedJobName,' ...
          'staticConfig,selectedInverseFilters);']);
end

%=========================================================================
% Plot the results
%=========================================================================
if (managerConfig.plotGraphs)
    eval(['[' mergedJobResultsVariableName ' resultInfo] = ' ...
          'DuctSimulationResultViewer(' mergedJobResultsVariableName ...
          ',selectedInverseFilters,simulationConfig,plotResultsConfig,staticConfig,runparams);']);
end
```

A.3.1.2 DuctSimulationSubmitJobsAndFetchResults.m

```
function mergedJobResults = DuctSimulationSubmitJobsAndFetchResults( ...
    mergedJobResults,submitJobsAndFetchResultsConfig,simulationConfig,runparams,mergedJobName, ...
    staticConfig,selectedInverseFilters)
submitFileSettings = submitJobsAndFetchResultsConfig.submitFileSettings;
%=========================================================================
% Initialise queuingComputer mat file to record what this computer has submitted to condor.
%=========================================================================
% Find queuing computer name (works on windows only)
if (submitJobsAndFetchResultsConfig.resubmitCondorJobs)
    if (ispc)
        queuingComputerName = getenv('COMPUTERNAME');
    else
        queuingComputerName = getenv('HOSTNAME');
    end
    queuingComputerName = strtrim(queuingComputerName);
    queuingComputerMatFilename = ['condorjobs_' queuingComputerName '.mat'];

    if (submitJobsAndFetchResultsConfig.resubmitCondorJobs)
        if exist(queuingComputerMatFilename)
            load(queuingComputerMatFilename);
            if (~exist('condorjobs') | ...
                ~isfield(condorjobs,'submitCount') | ...
                ~isfield(condorjobs.submitCount,mergedJobName))
                eval(['condorjobs.submitCount.' mergedJobName ' = 0;']);
            end
```

```
26          else
27              eval(['condorjobs.submitCount.' mergedJobName ' = 0;']);
28          end
29          eval(['condorjobs.submitCount.' mergedJobName ' = ' ...
30                  'condorjobs.submitCount.' mergedJobName ' + 1;']);
31          setSubmitCount = getfield(condorjobs.submitCount, mergedJobName);
32          save(queuingComputerMatFilename, 'condorjobs');
33
34          warning off MATLAB:MKDIR:DirectoryExists
35          mkdir(mergedJobName);
36          warning on MATLAB:MKDIR:DirectoryExists
37      end
38  end
39
40
41  %-------------------------------------------------------------------------------
42  % Initialise general variables
43  %-------------------------------------------------------------------------------
44  if (~submitJobsAndFetchResultsConfig.usePreviousFetchedResult)
45      clear mergedJobResults;
46  end
47
48  condorJobsRemainingCount = 0;
49
50  % Generate the string, '%0x' where x is (1 for 1-9 jobs, 2 for 10-99 jobs, etc..)
51  numberOfDigits = ceil(log10(...
52      length(staticConfig.ranges.frequencyCarrier) ...  % Place in here all the loop lengths
53      ));
54
55  % Create a job submission order variable
56  if submitJobsAndFetchResultsConfig.submitJobsInRandomOrder
57      jobOrder = randsample(length(staticConfig.ranges.frequencyCarrier),...
58                            length(staticConfig.ranges.frequencyCarrier));
59  else
60      jobOrder = 1:length(staticConfig.ranges.frequencyCarrier);
61  end
62
63  if (submitJobsAndFetchResultsConfig.usePreviousFetchedResult)
64      tst = ~exist('mergedJobResults');
65      if (tst)
66          mergedJobResults.resultsFetched = zeros(1,length(staticConfig.ranges.frequencyCarrier));
67          disp('The variable doesn''t exist, can''t use previously fetched results!');
68      else
69          disp('Using existing results to speed up resubmissions!');
70      end
71  end
72
73  waitbarHandle = MultiWaitBar([0 0],'Loading files for mergedJobResults');
74  for jobOrderIndex = 1:length(staticConfig.ranges.frequencyCarrier)
75      frequencyCarrierIndex = jobOrder(jobOrderIndex);
76
77      if (submitJobsAndFetchResultsConfig.usePreviousFetchedResult)
78          thisResultPreviouslyFetched = mergedJobResults.resultsFetched(frequencyCarrierIndex);
79      else
80          thisResultPreviouslyFetched = 0;
81      end
82
83      if ((~submitJobsAndFetchResultsConfig.usePreviousFetchedResult) ...
84          || (~thisResultPreviouslyFetched))
85          freqCarrier = staticConfig.ranges.frequencyCarrier(frequencyCarrierIndex);
86          jobIndex = frequencyCarrierIndex;
87          jobDirectoryName = sprintf([mergedJobName '/run%04d/'],jobIndex);
88          runparams.fc      = freqCarrier;
89          runparams.tofile = sprintf(['run%04d.mat'],jobIndex);
90
```

```
91          if (exist([jobDirectoryName runparams.tofile],'file'))
92              %-------------------------------------------------------------------------
93              % If the results exist in a mat file, them...
94              %-------------------------------------------------------------------------
95              load([jobDirectoryName runparams.tofile]);
96              resultsFetched(frequencyCarrierIndex) = 1;
97
98              % Add the data from loaded matfile, (in thisresults), to the mergedJobResultsVariable.
99              for filterIndex = 1:length(selectedInverseFilters.textKeys)
100                 fn = selectedInverseFilters.textKeys{filterIndex};
101                 for generalMeasureVariableIndex = ...
102                         1:length(staticConfig.results.generalMeasureVariableNames)
103                     generalMeasureVariableName = ...
104                         staticConfig.results.generalMeasureVariableNames{generalMeasureVariableIndex};
105
106                     eval(['mergedJobResults.' fn '.' generalMeasureVariableName ...
107                           '(:,:,frequencyCarrierIndex) = ' ...
108                           'thisresults.' fn '.' generalMeasureVariableName ';']);
109                 end
110
111                 eval(['tst = isfield(thisresults.' fn ',''skipping'');'])
112                 if tst
113                     eval(['mergedJobResults.' fn ...
114                           '.skipped(:,:,frequencyCarrierIndex) = thisresults.' fn '.skipping;']);
115                 else
116                     if selectedInverseFilters.betaVaries(filterIndex)
117                         betaLength = length(staticConfig.ranges.beta);
118                     else
119                         betaLength = 1;
120                     end
121                     eval(['mergedJobResults.' fn ...
122                           '.skipped(:,:,frequencyCarrierIndex) = -1*ones(betaLength,2);']);
123                 end
124
125                 for adaptiveFilterIndex = 1:length(staticConfig.results.adaptiveFilterKeys)
126                     adaptiveFilterKey = staticConfig.results.adaptiveFilterKeys{adaptiveFilterIndex};
127                     for adaptiveFilterMeasureIndex = ...
128                             1:length(staticConfig.results.adaptiveFilterMeasureKeys)
129
130                         adaptiveFilterMeasureKey = ...
131                             staticConfig.results.adaptiveFilterMeasureKeys{adaptiveFilterMeasureIndex};
132                         eval([ ...
133                             'mergedJobResults.' fn '.' ...
134                             adaptiveFilterKey adaptiveFilterMeasureKey ...
135                             '(:,:,:,frequencyCarrierIndex) = thisresults.' fn '.' ...
136                             adaptiveFilterKey adaptiveFilterMeasureKey ...
137                             ';']);
138                     end
139                 end
140                 eval(['mergedJobResults.' fn ...
141                       '.cputime(:,:,frequencyCarrierIndex) = thisresults.' fn '.cputime'';']);
142             end
143             mergedJobResults.totaltime(frequencyCarrierIndex) = thisresults.totaltime;
144             % mergedJobResults.cputime(frequencyCarrierIndex) = thisresults.cputime;
145         else
146             %-------------------------------------------------------------------------
147             % If the results don't exist, fake the results and resubmit (dependent on
148             % submitJobsAndFetchResultsConfig)
149             %-------------------------------------------------------------------------
150             resultsFetched(frequencyCarrierIndex) = 0;
151             condorJobsRemainingCount = condorJobsRemainingCount + 1;
152
153             % Fake the result
154             if (submitJobsAndFetchResultsConfig.fakeMissingResults && frequencyCarrierIndex > 2)
155                 for filterIndex = 1:length(selectedInverseFilters.textKeys)
```

```
156                     fn = selectedInverseFilters.textKeys{filterIndex};
157                     eval(['mergedJobResults.' fn '.total_std(:,:,frequencyCarrierIndex) = ' ...
158                         'mergedJobResults.' fn '.total_std(:,:,frequencyCarrierIndex-1);']);
159
160                     eval(['mergedJobResults.' fn '.gain(:,:,frequencyCarrierIndex)      = ' ...
161                         'mergedJobResults.' fn '.gain(:,:,frequencyCarrierIndex-1);']);
162
163                     eval(['mergedJobResults.' fn '.crosstalk(:,:,frequencyCarrierIndex) = ' ...
164                         'mergedJobResults.' fn '.crosstalk(:,:,frequencyCarrierIndex-1);']);
165                 end
166             end
167
168             % Submit or resubmit Condor jobs
169             if (submitJobsAndFetchResultsConfig.resubmitCondorJobs)
170                 disp(['SubmitCondorJob(jobIndex,' jobDirectoryName ...
171                     ',submitFileSettings,5,runparams']);
172                 condorJobId = SubmitCondorJob(jobIndex,jobDirectoryName, ...
173                                               submitFileSettings,5,runparams);
174
175                 load(queuingComputerMatFilename,'condorjobs');
176                 eval(['condorjobs.' mergedJobName '.nows(setSubmitCount,jobIndex)     = now();']);
177                 eval(['condorjobs.' mergedJobName '.started(setSubmitCount,jobIndex) = 1;']);
178                 eval(['condorjobs.' mergedJobName '.jobid(setSubmitCount,jobIndex) =condorJobId;']);
179                 save(queuingComputerMatFilename,'condorjobs');
180             end
181         end
182     end
183     MultiWaitBar([jobOrderIndex-condorJobsRemainingCount condorJobsRemainingCount] ...
184                  ./length(staticConfig.ranges.frequencyCarrier),waitbarHandle);
185 end
186 close(waitbarHandle)
187 fprintf('Have %d results, waiting on %d.\n',[sum(resultsFetched) sum(1-resultsFetched)]);
188 eval(['mergedJobResults.resultsFetched = resultsFetched;']);
```

A.3.1.3 DuctSimulationResultViewer.m

```
1  function [mergedJobResults resultInfo] = DuctSimulationResultViewer( ...
2      mergedJobResults,selectedInverseFilters,simulationConfig,plotResultsConfig,staticConfig,runparams)
3
4  %-------------------------------------------------------------------------------------------
5  % Determine useful variables
6  %-------------------------------------------------------------------------------------------
7
8  % Load the sampling frequency from the pre-measured results.
9  testIRFsMatVariables = load('/media/KFUSB2/KFUSB1-thesisSimulations/SubmitScript/testIRFs.mat');    % Contains IRFs, fs
10 freqSampling = testIRFsMatVariables.fs;
11
12 selectedBetaVariantFiltersIndex = find(selectedInverseFilters.betaVaries);
13 selectedBetaVariantFiltersCount = ...
14     length(selectedInverseFilters.textKeys(selectedBetaVariantFiltersIndex));
15
16 validFreqIndexes = find(mergedJobResults.resultsFetched);
17
18 % Create an index of all the integer frequencies.. This occurs when the value after the decimal
19 % resets, (i.e. goes from 9 to 0 as in 11.81 11.95 12.02)
20 decimalValue = ...
21     staticConfig.ranges.frequencyCarrier/1e3 - floor(staticConfig.ranges.frequencyCarrier/1e3);
22 integerKHzFrequencyIndex = find((decimalValue(2:end) - decimalValue(1:end-1)) < 0) + 1;
23
24 arrangementText = ['[' strtrim(sprintf('%1d ',runparams.IRFchannelidx)) '] => [1 2]'];
25
26 symbolLengthRangeCount = length(staticConfig.ranges.symbolLength);
27
```

```
28 symbolLength = runparams.symblen;
29
30 %------------------------------------------------------------------------------
31 % Calculate derived simulation results
32 %------------------------------------------------------------------------------
33 mergedJobResults = CondorSubmitterCalculateDerivedVariables(mergedJobResults, ...
34     selectedInverseFilters, ...
35     length(staticConfig.ranges.transmitterElements{simulationConfig.arrangementIndex}));
36
37 %------------------------------------------------------------------------------
38 % Plot the graphs
39 %------------------------------------------------------------------------------
40 figurePositionVariable = [ ...
41     (simulationConfig.symbolLengthIndex-1)*1280/symbolLengthRangeCount ...
42     50 ...
43     1280/symbolLengthRangeCount ...
44     904];
45
46 %                          x y w h
47 subplotGrid.position = plotResultsConfig.subplotGridPosition;
48 subplotGrid.spacing  = plotResultsConfig.subplotGridSpacing;
49 subplotGrid.size     = [length(staticConfig.ranges.symbolLength) 4];
50
51 subplotGrid.derived.xStep  = subplotGrid.position(3) / subplotGrid.size(1) ...
52     + subplotGrid.spacing(1)/2;
53 subplotGrid.derived.yStep  = subplotGrid.position(4) / subplotGrid.size(2) ...
54     + subplotGrid.spacing(2)/2;
55 subplotGrid.derived.width  = subplotGrid.position(3) / subplotGrid.size(1) ...
56     - subplotGrid.spacing(1)/2;
57 subplotGrid.derived.height = subplotGrid.position(4) / subplotGrid.size(2) ...
58     - subplotGrid.spacing(1)/2;
59
60 resultInfo.names = {
61     'Transmitter power without normalisation'                                ...
62     'Power at target receiver from transmission for target receiver'         ...
63     'Power at target receiver from transmission for target receiver (2)'     ...
64     'Power at target receiver from transmission for target receiver (3)'     ...
65     'Power at target receiver from transmission for cross-talk receiver'     ...
66     ...
67     'Average symbol amplitude'                                               ...
68     ...
69     'Ratio of target signal to cross-talk signal power'                      ...
70     ...
71     'Symbol error'                                                           ...
72     'Symbol error using MSE adaptive filter'                                 ...
73     'Symbol error using MSE adaptive filter with training (40 symbols)'      ...
74     'Symbol error using ZF  adaptive filter'                                 ...
75     'Symbol error using ZF adaptive filter with training (40 symbols)'       ...
76     'Estimated symbol error using standard deviation'                        ...
77     'Estimated symbol error using standard deviation with cross-talk'        ...
78     ...
79     'Increase in standard deviation required to get P_e=1/400'               ...
80     'Increase in standard deviation required to get P_e=1/400 with cross-talk' ...
81     ...
82     'Noise level required to get P_e=1/400'                                  ...
83     'Noise level required to get P_e=1/400 with cross-talk'                  ...
84             };
85
86 resultInfo.fieldVariable = {
87     '_dv.txRMS'                    ...
88     '_dv.rxRMS0'                   ...
89     '_dv.rxRMS1'                   ...
90     '_dv.rxRMS2'                   ...
91     '_dv.rxRMSCT'                  ...
92     ...
```

```
93      '_dv.symbolAmplitude'                   ...
94      ...
95      '_dv.ratioRXToCT'                       ...
96      ...
97      '.symbolerr'                            ...
98      '.msea_symbolerr'                       ...
99      '.mseat_symbolerr'                      ...
100     '.zfa_symbolerr'                        ...
101     '.zfat_symbolerr'                       ...
102     '_dv.symbolErrorFromStdDev'             ...
103     '_dv.symbolErrorFromStdDevCT'           ...
104     ...
105     '_dv.increaseOfStdFor1in400'            ...
106     '_dv.increaseOfStdFor1in400WithCT'      ...
107     ...
108     '_dv.noiseForStd400'                    ...
109     '_dv.noiseForStd400CT'                  ...
110                     };
111
112 resultInfo.units = {
113     sprintf('dB')                   ... % txpower
114     sprintf('dB')                   ...
115     sprintf('dB')                   ...
116     sprintf('dB')                   ...
117     sprintf('dB')                   ...
118     ...
119     sprintf('dB')                   ... %
120     ...
121     sprintf('dB')                   ... %
122     ...
123     sprintf('Symbols\nincorrect')   ... % symbolerr
124     sprintf('Symbols\nincorrect')   ... % msea_symbolerr
125     sprintf('Symbols\nincorrect')   ...
126     sprintf('Symbols\nincorrect')   ...
127     sprintf('Symbols\nincorrect')   ...
128     sprintf('Symbols\nincorrect')   ...
129     sprintf('Symbols\nincorrect')   ...
130     ...
131     sprintf('dB')                   ...
132     sprintf('dB')                   ...
133     ...
134     sprintf('dB')                   ...
135     sprintf('dB')                   ...
136             };
137
138 resultInfo.logAmplitudes = {
139     1 ...
140     1 ...
141     1 ...
142     1 ...
143     1 ...
144     ...
145     1 ...
146     ...
147     1 ...
148     ...
149     0 ...
150     0 ...
151     0 ...
152     0 ...
153     0 ...
154     0 ...
155     0 ...
156     ...
157     1 ...
```

```
158       1 ...
159       ...
160       1 ...
161       1 ...
162                         };
163
164   colorLimit1 = [-5      35];
165   resultInfo.colorLimits = {
166       [-70   -14] ... %-12.97 -16.47 -15.92
167       [-55      0] ... % -6.8   -4.5   -3.5   (WAS: 8.7     5.0     2.5)
168       [-55      0] ... %-15.2  -10.6  -10.4   (WAS: 0.4    -1.0    -4.4)
169       [-55      0] ... % -4.7   -2.4   -1.7   (WAS: 10.0     7.1     4.4)  (-30 was good)
170       [-55      0] ... %  36.0    21.1    17.7   (WAS: 51.7    30.7    23.8) (upper 15)
171       ...
172       [-55      5] ... %  1.2      3.0     3.3   (WAS: 16.8    12.6     9.3)
173       ...
174       colorLimit1 ... % 49.7     49.8    22.0   (???: 32.7    26.2    22.0)
175       ...
176       [ 0    120] ...
177       [ 0    120] ...
178       [ 0    120] ...
179       [ 0    120] ...
180       [ 0    120] ...
181       [ 0    120] ...
182       [ 0    120] ...
183       ...
184       [ 0     20] ... % 19.75  19.02  17.69
185       [ 0     20] ... % 16.75  12.24  7.31
186       ...
187       [-45   -9] ...
188       [-45   -9] ...
189                         };
190
191   colorMapJet = jet(512);
192
193   zeroPosition = (0-colorLimit1(1))/(colorLimit1(2) - colorLimit1(1));
194   noPoints = floor(size(colorMapJet,1)*zeroPosition/(1-zeroPosition));
195   colorMap1 = [[ ...
196       zeros(noPoints+1,1) ...
197       (0.3+(0.1*[0:noPoints]'./noPoints)) ...
198       (0.2+(0.8*[0:noPoints]'./noPoints))
199                         ]; colorMapJet];
200
201   colorMapDefault = colorMapJet;
202
203   resultInfo.colorMaps = {
204       colorMapDefault ...
205       colorMapDefault ...
206       colorMapDefault ...
207       colorMapDefault ...
208       colorMapDefault ...
209       ...
210       colorMapDefault ...
211       ...
212       colorMap1 ...
213       ...
214       colorMapDefault ...
215       colorMapDefault ...
216       colorMapDefault ...
217       colorMapDefault ...
218       colorMapDefault ...
219       colorMapDefault ...
220       colorMapDefault ...
221       ...
222       colorMapDefault ...
```

```
223     colorMapDefault ...
224     ...
225     colorMapDefault ...
226     colorMapDefault ...
227                     };
228
229 % Figure start count
230 fcs = 1 + (length(resultInfo.fieldVariable) + 1)*(simulationConfig.symbolLengthIndex-1);
231
232 if (plotResultsConfig.showMainResults)
233     for figureCount = 1:length(resultInfo.fieldVariable)
234         figureFieldVariable = resultInfo.fieldVariable{figureCount};
235         figure(figureCount)
236         % [basex basey width height]
237         set(gcf, ...
238             'DefaultAxesPosition',[.06 .10 1-2*.05 - 0.04 1-2*.08], ...
239             'name',resultInfo.names{figureCount}, ...
240             'position',[184 212 1062 713]);
241         set(gcf, ...
242             'PaperOrientation','landscape', ...
243             'PaperPosition',[[0.634517 0.634517 28.4084 19.715]]);
244
245
246         % (1) Find the limits of the data from all the results (to isoloate the max / min for the
247         % color limits)
248         filterData = [];
249         for filterIndex = 1:selectedBetaVariantFiltersCount;
250             fn = selectedInverseFilters.textKeys{selectedBetaVariantFiltersIndex(filterIndex)};
251             eval(['filterData = [filterData; ' ...
252                 'squeeze(mergedJobResults.' fn figureFieldVariable '(:,1,validFreqIndexes))];']);
253         end
254         clim = [min(20*log10(filterData(:))) max(20*log10(filterData(:)))];
255
256         % (2) Loop through filter types
257         for filterIndex = 1:selectedBetaVariantFiltersCount;
258             fn = selectedInverseFilters.textKeys{selectedBetaVariantFiltersIndex(filterIndex)};
259             receiverChannel = 1;
260
261             subplot('Position',[ ...
262                 subplotGrid.position(1) + subplotGrid.derived.xStep*(simulationConfig.symbolLengthIndex-1) ...
263                 1 - (subplotGrid.position(2) + subplotGrid.derived.yStep*(filterIndex)) ...
264                 subplotGrid.derived.width ...
265                 subplotGrid.derived.height ...
266                                 ]);
267
268             % Plot the image
269             eval(['filterData = mergedJobResults.' fn figureFieldVariable ';']);
270             if (resultInfo.logAmplitudes{figureCount})
271                 plotData = 20*log10([squeeze(filterData(:,receiverChannel,:))]);
272             else
273                 plotData = [squeeze(filterData(:,receiverChannel,:))];
274             end
275
276             % Set color range
277             if (length(resultInfo.colorLimits{figureCount}))
278                 nullPlotData = zeros(size(plotData));
279                 eval(['nullPlotData = squeeze(mergedJobResults.' fn '.skipped(:,1,:)) > 0;']);
280                 nullPlotData(:,find(mergedJobResults.resultsFetched == 0)) = 1;
281                 % make a line to indicate frequencies that aren't valid
282                 nullPlotData(:, find(...
283                     staticConfig.ranges.frequencyCarrier < freqSampling/symbolLength/2 ...
284                     )) = 1;
285                 MarkNullImageSC(plotData([1 1 1:size(plotData,1)],:), ...
286                                 resultInfo.colorLimits{figureCount}, ...
287                                 nullPlotData([1 1 1:size(plotData,1)],:), ...
```

```
288                         [1 1 1]*0.8, ...
289                         resultInfo.colorMaps{figureCount} ...
290                         );
291             else
292                 imagesc(plotData([1 1 1:size(plotData,1)],:));
293                 colormap(resultInfo.colorMaps{figureCount});
294             end
295             h = line([0 665],[3.5 3.5]);
296             set(h,'LineStyle','-','Color',[1 1 1])
297
298             h = line([0 665],[3.5 3.5]);
299             set(h,'LineStyle','--','Color',[0 0 0])
300
301             lastPlotAxes = gca;
302
303             % Column titles
304             currentFigurePosition = get(lastPlotAxes,'position');
305             if (filterIndex == 1)
306                 titleHandle = title(sprintf('%.0f baud',freqSampling/symbolLength));
307                 titlePosition = get(titleHandle,'position');
308                 set(titleHandle, ...
309                     'position',[titlePosition(1) -.3 1], ...
310                     'fontsize',10.5)
311             end
312
313             % Y Labels
314             if (simulationConfig.symbolLengthIndex == 1)
315                 set(lastPlotAxes, ...
316                     'YTick',2+[2:8:length(staticConfig.ranges.beta)]);
317                 set(gca,'Fontsize',10);
318                 set(lastPlotAxes, ...
319                     'YTickLabel',(staticConfig.ranges.beta(-2+get(lastPlotAxes,'YTick'))));
320
321                 yLabelHandle = ...
322                     ylabel(sprintf('%s\n\\kappa', ...
323                                    selectedInverseFilters.abbreviations{ ...
324                                        selectedBetaVariantFiltersIndex(filterIndex)} ...
325                                    ));
326                 ylp = get(yLabelHandle,'position');
327                 % 1.5 to adjust top after changing VerticalAlignment from middle to top.
328                 set(yLabelHandle, ...
329                     'FontSize',10.5, ...
330                     'VerticalAlignment','top', ...
331                     'Position',ylp.*[1.35 1 1]);
332                 lastYLabelHandle = yLabelHandle;
333             else
334                 set(lastPlotAxes,'YTick',[]);
335                 set(lastPlotAxes,'YTickLabel',[]);
336             end
337
338             % X Labels, (Time Reversal is used for the x-labels)
339             set(lastPlotAxes,'XTick',[]);
340             set(lastPlotAxes,'XTickLabel',[]);
341
342             % Plot time reversal data
343             if (filterIndex == selectedBetaVariantFiltersCount)
344                 currentFigurePosition = get(lastPlotAxes,'position');
345                 axes('position', ...
346                      plotResultsConfig.timeReversalYPosition*[0 1 0 0; 0 0 0 1] + ...
347                      currentFigurePosition.*[1 0 1 0]);
348                 eval(['filterDataTR = mergedJobResults.tr' figureFieldVariable ';']);
349                 if (resultInfo.logAmplitudes{figureCount})
350                     plotDataTR = 20*log10([squeeze(filterDataTR(:,receiverChannel,:))]');
351                 else
352                     plotDataTR = [squeeze(filterDataTR(:,receiverChannel,:))]';
```

```
353             end
354             if (length(resultInfo.colorLimits{figureCount}))
355                 nullPlotDataTR = zeros(size(plotDataTR));
356                 nullPlotDataTR(find(mergedJobResults.resultsFetched == 0)) = 1;
357                 nullPlotDataTR(find(squeeze(mergedJobResults.tr.skipped(:,1,:)) > 0) ) = 1;
358
359                 % make a line to indicate frequencies that aren't valid
360                 nullPlotDataTR(find( ...
361                     staticConfig.ranges.frequencyCarrier < freqSampling/symbolLength/2 ...
362                     )) = 1;
363                 MarkNullImageSC(plotDataTR, ...
364                                 resultInfo.colorLimits{figureCount}, ...
365                                 nullPlotDataTR, ...
366                                 [1 1 1]*0.8, ...
367                                 resultInfo.colorMaps{figureCount});
368             else
369                 imagesc(plotDataTR);
370                 colormap(resultInfo.colorMaps{figureCount});
371             end
372
373             set(gca,'YTick',[]);
374             set(gca,'YTickLabel', []);
375             set(gca,'XTick',integerKHzFrequencyIndex(1:2:end));
376             set(gca, ...
377                 'XTickLabel', floor(...
378                     staticConfig.ranges.frequencyCarrier(integerKHzFrequencyIndex(1:2:end)) ...
379                     /1000));
380             xLabelHandle = xlabel('Carrier Frequency (kHz)');
381             set(xLabelHandle,'fontsize',10.5);
382
383             if (simulationConfig.symbolLengthIndex == 1)
384                 yLabelHandle = ylabel('TR');
385                 ylp  = get(yLabelHandle,'position');
386                 lylp = get(lastYLabelHandle,'position');
387                 set(yLabelHandle, ...
388                     'FontSize',10.5, ...
389                     'VerticalAlignment','top', ...
390                     'position',ylp.*[0 1 1] + lylp.*[1 0 0] ...
391                     );
392             end
393         end
394
395         % Draw colorbar for the last filter
396         if (length(findobj(gcf,'tag','figure_color_bar')) == 0 ...
397             && length(resultInfo.colorLimits{figureCount}) ...
398             && filterIndex == selectedBetaVariantFiltersCount)
399             lastAxesPosition = get(lastPlotAxes,'position');
400             colorbarAxes = axes;
401             % column of data
402             r = resultInfo.colorLimits{figureCount};
403             colorBarPlotData = [(r(1) + (r(2) - r(1))*[0:512]'./512)];
404             % MATLAB BUG: print -dpdf breaks with only one
405             colorBarPlotData2 = [colorBarPlotData colorBarPlotData];
406             MarkNullImageSC(colorBarPlotData2, ...
407                             resultInfo.colorLimits{figureCount}, ...
408                             zeros(size(colorBarPlotData2)), ...
409                             [1 1 1]*0.8, ...
410                             resultInfo.colorMaps{figureCount}, ...
411                             1,colorBarPlotData);
412
413             set(colorbarAxes, ...
414                 'YDir','normal',...
415                 'XTick',[],...
416                 'XTickLabel', [],...
417                 'tag','figure_color_bar', ...
```

```
418                     'YAxisLocation', 'right', ...
419                     'FontSize',10, ...
420                     'position',plotResultsConfig.colorbarXPosition*[1 0 0 0; 0 0 1 0] + ...
421                     [0 ...
422                      plotResultsConfig.timeReversalYPosition(1) ...
423                      0 ...
424                      (1 + subplotGrid.derived.height - subplotGrid.derived.yStep ...
425                       - (plotResultsConfig.timeReversalYPosition(1) + subplotGrid.position(2))) ...
426                     ]);
427                 title(resultInfo.units{figureCount})
428             end
429         end
430
431         % Total figure title
432         if (simulationConfig.symbolLengthIndex == 1)
433             ah = annotation('textbox',plotResultsConfig.annotationPosition, ...
434                     'String',sprintf('%s : %s',arrangementText,resultInfo.names{figureCount}), ...
435                     'LineStyle','none', ...
436                     'LineWidth',0, ...
437                     'HorizontalAlignment','center', ...
438                     'VerticalAlignment','baseline', ...
439                     'FontSize',14);
440         end
441
442     end
443 end
444
445 if (plotResultsConfig.showTXPowerVSPeak)
446     figureCount = figureCount+1;
447     selectedFilterCount = length(selectedInverseFilters.textKeys);
448     % Histogram of the spread of ratios for txpeak txpower
449     figure(fcs + figureCount - 1);
450     set(gcf,'Position',figurePositionVariable);
451     set(gcf,'PaperPosition',[0.634517 0.634517 19.715 28.4084]);
452     subplot(selectedFilterCount,2,1)
453     for filterIndex = 1:selectedFilterCount;
454         fn = selectedInverseFilters.textKeys{filterIndex};
455         eval(['tpeak = squeeze(mergedJobResults.' fn '.txpeak);']);
456         eval(['tpower = squeeze(mergedJobResults.' fn '.txpower);']);
457         subplot(selectedFilterCount,2,(filterIndex-1)*2 + 1)
458         h = plot(tpower(:),tpeak(:),'.');
459         set(h,'MarkerSize',1);
460         axis([0 .06 0 .4]);
461         title([selectedInverseFilters.names{filterIndex}]);
462         subplot(selectedFilterCount,2,(filterIndex-1)*2 + 2)
463         warning off MATLAB:dividebyzero
464         hist(tpeak(:)./tpower(:),[1.3:.02:12]);
465         warning on MATLAB:dividebyzero
466         ax = axis; axis([1.3 12 ax(3:4)]);
467     end
468     xlabel(sprintf('n_{symb}=%d, resulting in f_{symb} = %.3f', ...
469                 symbolLength,freqSampling/symbolLength));
470 end
471
472 if (plotResultsConfig.showCondorTime)
473     figureCondorTimeHandle = findobj(0,'Tag','CondorTime');
474     if (~length(figureCondorTimeHandle))
475         figureCondorTimeHandle = figure(40);
476         set(figureCondorTimeHandle,'Tag','CondorTime');
477     else
478         figure(figureCondorTimeHandle)
479     end
480     totalTimesSorted = sort(mergedJobResults.totaltime);
481     hold on;
482     plot(totalTimesSorted./60./60,'x');
```

```
483     ylabel('Total execution time (hours)');
484 end
```

A.3.2 Functions for the Condor submitter script

A.3.2.1 SubmitCondorJob.m

```
1  function condorJobId = SubmitCondorJob( ...
2      jobIndex,jobDirectoryName,submitFileSettings,verbosityLevel,runparams)
3  % USAGE:  condorJobId = SubmitCondorJob( ...
4  %    jobIndex,jobDirectoryName,submitFileSettings,verbosityLevel,runparams)
5  %
6  % This function is used to create a condor submission file, along with a parameter file in order
7  % to run condor jobs, and then submits the job to Condor. The parameters are as follows:
8  %
9  %    jobIndex           - an integer value that represents the index of the job.
10 %                         (i.e. the 5 in '5 of 20')
11 %
12 %    jobDirectoryName   - is the directory to submit the job in.
13 %
14 %    submitFileSettings - is a structure containing information used when forming the condor
15 %                         submit file.  The following fields must be declared:
16 %
17 %        .environmentPath         - A string of the environmental search path
18 %        .notify                  - An integer to determine the inclusion of "notification = never"
19 %        .executable              - A string of the file to execute
20 %        .additionalTransferFiles - A string of the extra files to tranfer, (must lead the string
21 %                                   with a comma (i.e. ',..\\foobar.mat')
22 %
23 %    verbosityLevel     -  is an integer to determine the level of messages to display:
24 %          0 - No messages, except error messages
25 %          1 - starting / finishing condor
26 %              job successfully submitted
27 %          2 - creation of runsettings.mat
28 %          3 - creation of submit file
29 %          4 - directory traversal
30 %
31 %    runparams          - is a structure that contains a set of parameters that is written to the
32 %                         file, runparams.mat.
33 %
34 %    condorJobId        - The job ID given by Condor, (used for tracking purposes)
35 %
36 % Usage Notes:
37 %
38 %    Due to the fact that condor is bad at transferring files, it is best to use the extrainfiles as
39 %    little as possible.  Ideally one should map a network drive, and place the matlab executable in
40 %    this directory, along with any common .mat.  Consider if you have a .mat files of only 100k and
41 %    you are doing a job consisting of 10000 runs, you file transfer will result in 1 Gb of file
42 %    transfer, and disk usage!
43
44 % File revision history:
45 %
46 %   2010-04, Pierre Dumuid
47 %     * Cleaned up for presentation in his thesis.
48 %
49 %   2006-08, Pierre Dumuid
50 %     * Modified to use params instead of a changing list of arguments.
51 %     * Implemented structures and nicer looking varable names.
52 %     * Made the help information better.
53 %
54 %   2006-0? Carl Olsard
55 %     * Adapted for the GA Toolbox for  use in an asynchronous manner with the Condor pool.
```

```
%
%   ????-?? Rick Morgans
%     * developed the code for the AFOSR project.
%

VerbosePrintString(verbosityLevel,1,'Starting Condor_submit.\n')

%---------------------------------------------------------------------------
% Clear up previously logs, errors, and out files
%---------------------------------------------------------------------------
warning off
if ispc
    try
        delete([jobDirectoryName,'\*.log']);
    end
    try
        delete([jobDirectoryName,'\*.err']);
    end
    try
        delete([jobDirectoryName,'\*.out']);
    end
else
    try
        delete([jobDirectoryName,'/*.log']);
    end
    try
        delete([jobDirectoryName,'/*.err']);
    end
    try
        delete([jobDirectoryName,'/*.out']);
    end
end
warning on

%---------------------------------------------------------------------------
% Create the directory to perform the work in
%---------------------------------------------------------------------------
warning off MATLAB:MKDIR:DirectoryExists
mkdir(jobDirectoryName);
warning on  MATLAB:MKDIR:DirectoryExists

%---------------------------------------------------------------------------
% Create the MATLAB parameter file
%---------------------------------------------------------------------------
VerbosePrintString(verbosityLevel,2,'Creating runparams.mat ...\n');
tic;
save([jobDirectoryName '/runparams.mat'],'runparams');
VerbosePrintString(verbosityLevel,2,sprintf(' done (took %.2f seconds)\n',toc));

%---------------------------------------------------------------------------
% Create the Condor submit file
%---------------------------------------------------------------------------
VerbosePrintString(verbosityLevel,3,'Creating submit file ...\n');
tic;

submitFileName = sprintf('submitFile%04d.sub',jobIndex);
submitFileHandle = fopen([jobDirectoryName '/' submitFileName],'w');

fprintf(submitFileHandle,'universe = vanilla\n');
fprintf(submitFileHandle,['environment = path=' submitFileSettings.environmentPath '\n']);
fprintf(submitFileHandle,[ ...
    'requirements = ' ...
    '(Arch == "INTEL") && ((OpSys == "WINNT50") || (OpSys == "WINNT51")) && (Disk > 260000) \n']);
%fprintf(submitFileHandle,'rank = Machine == "catspc001.cats.adelaide.edu.au"\n');
%fprintf(submitFileHandle, [ ...
```

```
%     'requirements = (OpSys == "WINNT51") && (Subnet != "129.127.239")\n');
if ~submitFileSettings.notify
    fprintf(submitFileHandle,'notification = never\n');
end
fprintf(submitFileHandle,'TRANSFER_FILES = ALWAYS \n');
fprintf(submitFileHandle,'should_transfer_files = YES\n');
fprintf(submitFileHandle,'when_to_transfer_output = ON_EXIT\n');
fprintf(submitFileHandle,['executable = ' submitFileSettings.executable ' \n']);
fprintf(submitFileHandle,[ ...
    'transfer_input_files = runparams.mat' submitFileSettings.additionalTransferFiles '\n']);
%fprintf(submitFileHandle,'TRANSFER_OUTPUT_FILES = out.mat,success.sub,failed.txt\n');
fprintf(submitFileHandle,'output = thisjob$(Cluster).out\n');
fprintf(submitFileHandle,'error = thisjob$(Cluster).err\n');
fprintf(submitFileHandle,'log = thisjob$(Cluster).log\n');
fprintf(submitFileHandle,'initialdir = .\n');
fprintf(submitFileHandle,'copy_to_spool = false \n');
fprintf(submitFileHandle,'on_exit_remove = (ExitCode != 1)\n');
fprintf(submitFileHandle,'queue %d\n',1);
fclose(submitFileHandle);
VerbosePrintString(verbosityLevel,3,sprintf('done (%.2f seconds)\n',toc));

%-------------------------------------------------------------------------------
% Perform the condor submission
%-------------------------------------------------------------------------------
VerbosePrintString(verbosityLevel,4,'Entering directory...\n');

% Save the current directory so that we can return to it later.
originalDirectory = pwd;

% Enter the directory containing the condor job files.
eval(['cd ',jobDirectoryName]);

% Try using a dos prompt to submit the condor job.
dosSubmitCommandString = [ 'condor_submit -d ' submitFileName ' > submitStdOut.txt' ];
errorCode = 1;
errorText = '';
while errorCode ~= 0
    % If there is an error in the submission, continue to try to resubmitting the job.
    VerbosePrintString(verbosityLevel,1,sprintf('Submitting job "%s" ... ',submitFileName));
    tic
    [errorCode,errorText] = dos(dosSubmitCommandString);
    if errorCode ~= 0
        toc
        pause(5);
        VerbosePrintString(verbosityLevel,1,' failed\n');
        VerbosePrintString(verbosityLevel,0, ...
                           ['#### ',datestr(now),' Had to resubmit job ' jobDirectoryName ...
                            '. The error was: \n']);
        VerbosePrintString(verbosityLevel,0,['ERROR TEXT=',errorText '\n']);
    else
        VerbosePrintString(verbosityLevel,1,sprintf('success! (took %.2f seconds)\n',toc));
        condorJobId = 0;
        fh = fopen('submitStdOut.txt');
        while 1
            tline = fgetl(fh);
            if ~ischar(tline), break; end
            [tok mat] = regexp(tline,'submitted to cluster ([0-9]*).','tokens','match');
            if length(tok);
                condorJobId = str2double(tok{1});
                break;
            end
        end
        fclose(fh);
    end
end
```

```
186
187 VerbosePrintString(verbosityLevel,4,'Leaving directory...\n');
188 eval(['cd ' originalDirectory ';']);
189
190 VerbosePrintString(verbosityLevel,1,'Condor_submit finished\n');
191
192
193 function VerbosePrintString(verbosityLevel,requiredVerbosityLevel,string)
194     if (verbosityLevel >= requiredVerbosityLevel)
195         fprintf('%s',string);
196     end
```

A.3.3 The Condor job script

A.3.3.1 The Condor job executor: CondorJobExecutor.bat

```
1  @echo off
2
3  REM This script is executed on each condor computer to mount a network drive, and then run the
4  REM MATLAB compiled program, CondorJobMainScript.exe.
5
6  REM Copy the file from the Network drive to the local drive
7
8  path=C:\WINNT\system32;C:\WINDOWS\system32;D:\temp;%PATH
9  echo path = %PATH%
10
11 echo ==== DEBUG-START: Listing all connected network drives:
12 net use
13 echo ==== DEBUG: Show All Mounts
14 mountvol
15 echo ==== DEBUG: Parameter passed is
16 echo %1
17 echo ==== DEBUG-END
18
19 echo ==== Trying different mount points
20 set MYDIR=\pmdumuid\test2
21
22 echo ==== attempt to connect to ts1.cats.adelaide.edu.au (IP=\\129.127.236.8)
23 for %%d in (B: E: F: G: H: I: J: K: L: M: N: O: P: Q: R: S: T: U:) do (
24   echo Attempt Drive, %%d
25   echo    1. Attempt to unmount drive, %%d
26   mountvol %%d /D
27   echo    2. Attempting to disconnect drive, %%d
28   net use %%d /d
29   echo    3. Attempting to mount using drive, %%d
30   net use %%d \\129.127.236.8\condor smbmountpassword /USER:condor /PERSISTENT:NO
31   if %ERRORLEVEL% LEQ 0 (
32     set td=%%d
33     goto :mountokay
34   )
35 )
36
37 exit /B 1
38
39 :mountokay
40 echo Mount successful!, the drive is %td%
41 echo %td% > mountedDriveLetter.txt
42 set PATH=%PATH%;%td%;%td%\matlab6_lib;%td%\matlab6_lib\bin\win32;%td%\matlab7_lib\v71\runtime\win32
43 set PATH=%PATH%;%td%%MYDIR%
44
45 echo The path is: %PATH%
46 echo The contents of %td%%MYDIR% is:
```

```
47 dir %td%\%MYDIR%
48
49 echo ==== Run the Main script
50 %td%\%MYDIR%\CondorJobMainScript.exe > doitout.txt
51
52 echo ==== finished calculation
53 if NOT exist run???.mat (
54   echo %computername% failed > failed.txt
55   exit /B 1
56 )
57
58 echo ==== DEBUG-START2: Path
59 path
60 echo ==== DEBUG: Directory Listing
61 dir .
62 echo ==== DEBUG-END
63
64 echo ==== Unmounting the drive, %td%
65 net use %td% /delete
66 echo === FINISHED
67
68 exit /B 0
```

A.3.3.2 The main Condor job: CondorJobMainScript.m

```
1  function CondorJobMainScript()
2  % USAGE: CondorJobMainScript()
3  %
4  %  This function is used as part of a simulation conducted that forms part of the thesis for Pierre
5  %  Dumuid. This script is designed to be compiled to an executable file, and serves as a main
6  %  control function for the simulation. The simulation consists of:
7  %
8  %   * Loading various control parameters from runparams.mat specific to the local execution of
9  %     this script.
10 %
11 %   * Loading the impulse response, and digital sequence located generally on a mapped network
12 %     drive, specified by the variable, runparams.matpath.
13 %
14 %   * Isolate the channels in the impulse response.
15 %
16 %   * Create a base-band modulated signal
17 %
18 %   * Run the simulation of the channel.
19 %
20
21     load runparams.mat                          % runparams.fc
22                                                 %           .betarange
23                                                 %           .symblen
24                                                 %           .tofile
25                                                 %           .IRFchannelidx
26     load([runparams.matpath 'testIRFs.mat']);   % IRFs
27                                                 % fs
28     load([runparams.matpath 'bitseq2.mat']);    % bitseq
29
30     % Isolate the impulse responses according to the desired indexing.
31     channelIRFs = IRFs(:,:,runparams.IRFchannelidx);
32
33     % Rename some variables (since Variable Renaming)
34     % testIRFs.mat   contains variable called 'fs'     --> freqSampling
35     % bitseq2.mat    contains variable called 'bitseq' --> digitalSequence
36     % runparams.mat contains a number of variables that are mapped to
37     %                jobOptions
38
```

```
39      freqSampling                = fs;
40      digitalSequence             = bitseq;
41      jobOptions.boolTestFilter   = runparams.runopts.do;
42      freqCarrier                 = runparams.fc;
43      betaRange                   = runparams.betarange;
44      symbolLength                = runparams.symblen;
45
46      txSignalInfo = CondorJobCreateModulatedSignal(symbolLength,freqSampling,digitalSequence);
47      thisresults  = CondorJobRunFilterTests(channelIRFs,freqSampling,freqCarrier,betaRange, ...
48                                             symbolLength,txSignalInfo,jobOptions);
49      thisresults.runparams = runparams;
50      save(runparams.tofile,'thisresults');
51  end
```

A.3.3.3 CondorJobCreateModulatedSignal.m

```
1   function txSignalInfo = CondorJobCreateModulatedSignal(symbolLength,freqSampling,digitalSequence)
2   % USAGE: txSignalInfo = CondorJobCreateModulatedSignal(symbolLength,freqSampling,digitalSequence)
3   %
4   % This function is used to convert the digital sequence, digitalSequence to a PSK modulated
5   % base-band signal.
6   %
7
8   fprintf('CondorJobCreateModulatedSignal: Make up a transmission signal\n');
9   fprintf('  Step 1: Map binary stream to complex symbols (PSK)\n');
10  modulationSettings.k        = 2;
11  modulationSettings.addphase = 0;
12  complexSymbols = BitSequenceToComplexSequence(digitalSequence,'PSK',modulationSettings);
13
14  fprintf('  Step 2: Prepend symbols with a lead in sequence (used for synchronisation\n');
15  leadInSequence = [1 zeros(1,5) 1 zeros(1,5)];
16  complexSymbols = [leadInSequence complexSymbols];
17
18  fprintf('  Step 3: Create the spectral shaping filters\n');
19  rCFLength = 2048 * 6;
20  rCFLength = ceil(rCFLength/symbolLength)*symbolLength;
21  truncFLength = 2048;
22  truncFLength = ceil(truncFLength/symbolLength)*symbolLength;
23
24  % Create the Raised Co-sine spectral shaping filters.
25  raisedCosineSqrtFilter = real(fftshift(ifft(sqrt( ...
26      RaisedCosineFrequencySpectrum([1:rCFLength/2 -rCFLength/2+1:0]*freqSampling/rCFLength, ...
27                                    symbolLength/freqSampling, ...
28                                    0.2) ...
29      ))));
30  raisedCosineFilter     = real(fftshift(ifft( ...
31      RaisedCosineFrequencySpectrum([1:rCFLength/2 -rCFLength/2+1:0]*freqSampling/rCFLength, ...
32                                    symbolLength/freqSampling, ...
33                                    0.2) ...
34      )));
35
36  % Truncate the filters to truncFLength
37  [dummy peakIndex] = max(raisedCosineSqrtFilter);
38  raisedCosineSqrtFilter = raisedCosineSqrtFilter(peakIndex + [-truncFLength:truncFLength-1]);
39  raisedCosineFilter     = raisedCosineFilter(peakIndex + [-truncFLength:truncFLength-1]);
40
41  % Normalise the filter co-efficients
42  raisedCosineFilter     = raisedCosineFilter     /sum(raisedCosineFilter);
43  raisedCosineSqrtFilter = raisedCosineSqrtFilter/sum(raisedCosineSqrtFilter);
44
45  % Truncate at -80 dB
46  raisedCosineFilter     = CentralPeakSignalTrim(raisedCosineFilter     ,-80,symbolLength);
47  raisedCosineSqrtFilter = CentralPeakSignalTrim(raisedCosineSqrtFilter,-80,symbolLength);
```

```
48
49 fprintf('  Step 4: Convert the symbol stream into the base-band stream.\n');
50 txSignalInfo.PSK.transmitSignalBB      = ComplexSequenceToSignal(complexSymbols, ...
51                                                 raisedCosineSqrtFilter,symbolLength);
52 txSignalInfo.PSK.signalBBRaisedCosine = ComplexSequenceToSignal(complexSymbols, ...
53                                                 raisedCosineFilter      ,symbolLength);
54
55 % Second channel is not transmitted.
56 txSignalInfo.PSK.transmitSignalBB(1:size(txSignalInfo.PSK.transmitSignalBB,1),2) ...
57     = zeros(1,size(txSignalInfo.PSK.transmitSignalBB,1));
58 fprintf('\n');
59
60 txSignalInfo.PSK.leadInSequence         = leadInSequence;
61 txSignalInfo.PSK.leadInSequenceLength = length(leadInSequence);
62 txSignalInfo.PSK.complexSymbols         = complexSymbols;
63 txSignalInfo.PSK.modulationSettings    = modulationSettings;
64 txSignalInfo.digitalSequence            = digitalSequence;
```

A.3.3.4 CondorJobRunFilterTests.m

```
1  function thisResults = CondorJobRunFilterTests(channelIRFs,freqSampling,freqCarrier,betaRange, ...
2                                                  symbolLength,txSignalInfo,jobOptions)
3  % USAGE: thisResults = CondorJobRunFilterTests(channelIRFs,freqSampling,freqCarrier,betaRange, ...
4  %                                               symbolLength,txSignalInfo,jobOptions)
5  %
6  % This function is used to run some test for a digital communication defined by txSignalInfo with
7  % using specific inverse filter designs as developed as part of the thesis by Pierre Dumuid. A
8  % parameter involved in the inverse filters is varied according to the range of values in
9  % betaRange, and specific options for this set of test is given in the variable, jobOptions.
10 %
11 tic
12
13 %=========================================================================
14 % Various configuration parameters
15 %=========================================================================
16 % These are extra options that have been hard-coded, and could be moved into the variable,
17 % jobOptions at a later time.
18
19 config.downSampleForFilterDesign = 1;
20 config.testFilters.nof            = 0;
21 config.runRLSAAlgorithm           = 0;
22 config.targetReceiverRange        = [1]; % Receiver locations to test
23 config.crosstalkReceiverRange     = [2]; % Corresponding crosstalk receivers to measure
24 config.fractionalSpacing          = 2;
25 config.performAdaptiveFiltering  = 1;
26 config.deltaSteps                 = [0  6e-3 4e-4 0];
27 % The channelIRFs were truncated to 35 ms.. shouldn't need a filter to span more than this!  and the
28 % RLS algorithm was set to use filters with taps length 0.15 of this length as it was REALLY slow!
29 config.adaptiveFilterLength              = floor(0.5*36e-3/(symbolLength/freqSampling));
30 config.adaptiveFeedbackFilterLength     = floor(0.5*36e-3/(symbolLength/freqSampling));
31 config.adaptiveRLSFilterLength          = floor(0.15 * config.adaptiveFilterLength);
32 config.adaptiveRLSFeedbackFilterLength = floor(0.15 * config.adaptiveFeedbackFilterLength);
33 config.filterPeakIndex                  = config.adaptiveFilterLength;
34 config.examineCrossTalk                 = 1;
35 config.examineTransmitSignal            = 1;
36
37 fprintf('CondorJobRunFilterTests: Run Simulations\n');
38 %=========================================================================
39 % Convert channel IRF's to baseband
40 %=========================================================================
41 fprintf('  STEP 1: Converting the channel IRFs to baseband\n');
42 % Truncate IRFs to 2048.
43 channelIRFLength = min(2048,size(channelIRFs,1));
```

```
44
45 channelIRFsBB = PassbandToBaseband(channelIRFs(1:channelIRFLength,:,:),freqSampling,freqCarrier);
46
47 %=========================================================================
48 % Convert the channel IRF's to a lower sampling rate
49 %=========================================================================
50 fprintf('  STEP 2: Down-sample channels to lower sampling rate\n');
51 if (config.downSampleForFilterDesign)
52     % Work with 5 samples / symbol rate:
53     freqSymbol = freqSampling/symbolLength;
54     freqSamplingBB = ceil(5*freqSymbol);
55     if (freqSamplingBB > freqSampling)
56         freqSamplingBB = freqSampling;
57     end
58     channelIRFsBBLS = ResampleIRFs( ...
59         [channelIRFsBB; ...
60          zeros(floor(2*symbolLength), size(channelIRFsBB,2), size(channelIRFsBB,3))], ...
61         freqSamplingBB,freqSampling);
62 else
63     channelIRFsBBLS = channelIRFsBB;
64 end
65 thisResults.finish1 = toc;
66
67 %=========================================================================
68 % Iterate through various values of beta
69 %=========================================================================
70 fprintf('  STEP 3: Iterate through the beta range\n');
71 tocStartBetaLoop = toc;
72 for betaIdx = 1:length(betaRange)
73     thisBeta = betaRange(betaIdx);
74     fprintf('-------------------------------------------------------------\n');
75     fprintf('BETA Loop %d of %d (beta = %.3f freqCarrier = %.3f )\n', ...
76             betaIdx,length(betaRange),thisBeta,freqCarrier);
77     fprintf('-------------------------------------------------------------\n');
78
79     %=====================================================================
80     % Design the inverse filters at baseband and the lower sampling rate
81     %=====================================================================
82     fprintf(' (a) Create Filters\n');
83     clear filterInfo.textKeys filterInfo.fullNames
84     clear filterData
85
86     filterIdx = 0;
87
88     % The filters for "no filtering", time reversal and are independent of beta, and are only
89     % calculated for the first beta value.
90     if (betaIdx == 1)
91         if (config.testFilters.nof)
92             filterIdx = filterIdx + 1;
93             filterInfo.textKeys{filterIdx}  = 'nof';
94             filterInfo.fullNames{filterIdx} = 'No Filtering';
95             fprintf('  * %s\n',filterInfo.fullNames{filterIdx});
96             cputimeBeforeFilterDesign = cputime;
97             filterData.nof.inverseIRFsBBLS     = CreateInverseFilter(channelIRFsBBLS, 'No Filtering');
98             filterData.nof.receiverFilterBBLS = ones(1,size(channelIRFs,2));
99             filterData.nof.cputimeFilterDesign = cputime - cputimeBeforeFilterDesign;
100        end
101
102        if (jobOptions.boolTestFilter.tr)
103            filterIdx = filterIdx + 1;
104            filterInfo.textKeys{filterIdx}  = 'tr';
105            filterInfo.fullNames{filterIdx} = 'Time Reversal';
106            fprintf('  * %s\n',filterInfo.fullNames{filterIdx});
107            cputimeBeforeFilterDesign = cputime;
108            filterData.tr.inverseIRFsBBLS     = CreateInverseFilter(channelIRFsBBLS, 'TimeReversal');
```

```
109             filterData.tr.receiverFilterBBLS = ones(1,size(channelIRFs,2));
110             filterData.tr.cputimeFilterDesign = cputime - cputimeBeforeFilterDesign;
111         end
112     end
113
114     if (jobOptions.boolTestFilter.mil2s)
115         filterIdx = filterIdx + 1;
116         filterSettings.mil2s.beta = thisBeta;
117         filterInfo.textKeys{filterIdx}  = 'mil2s';
118         filterInfo.fullNames{filterIdx} = 'Milica Stojanovic (2 sided)';
119         fprintf('  * %s\n',filterInfo.fullNames{filterIdx});
120         cputimeBeforeFilterDesign = cputime;
121         [filterData.mil2s.inverseIRFsBBLS filterData.mil2s.filterDesignExtraInfo] = ...
122             CreateInverseFilter(channelIRFsBBLS,'Milica: Two-Side Filter',filterSettings.mil2s);
123         filterData.mil2s.receiverFilterBBLS = filterData.mil2s.filterDesignExtraInfo.rxFilter;
124         filterData.mil2s.cputimeFilterDesign = cputime - cputimeBeforeFilterDesign;
125     end
126
127     if (jobOptions.boolTestFilter.if_path)
128         filterIdx = filterIdx + 1;
129         filterSettings.if_path.beta = thisBeta;
130         filterInfo.textKeys{filterIdx}  = 'if_path';
131         filterInfo.fullNames{filterIdx} = 'Tikhonov IF: Path';
132         fprintf('  * %s\n',filterInfo.fullNames{filterIdx});
133         cputimeBeforeFilterDesign = cputime;
134         filterData.if_path.inverseIRFsBBLS = ...
135             CreateInverseFilter(channelIRFsBBLS,'Tikhonov IF: Path',filterSettings.if_path);
136         filterData.if_path.receiverFilterBBLS = ones(1,size(channelIRFs,2));
137         filterData.if_path.cputimeFilterDesign = cputime - cputimeBeforeFilterDesign;
138     end
139
140     if (jobOptions.boolTestFilter.if_chan)
141         filterIdx = filterIdx + 1;
142         filterSettings.if_chan.beta = thisBeta;
143         filterInfo.textKeys{filterIdx}  = 'if_chan';
144         filterInfo.fullNames{filterIdx} = 'Tikhonov IF: Channel';
145         fprintf('  * %s\n',filterInfo.fullNames{filterIdx});
146         cputimeBeforeFilterDesign = cputime;
147         filterData.if_chan.inverseIRFsBBLS = ...
148             CreateInverseFilter(channelIRFsBBLS,'Tikhonov IF: Channel',filterSettings.if_chan);
149         filterData.if_chan.receiverFilterBBLS = ones(1,size(channelIRFs,2));
150         filterData.if_chan.cputimeFilterDesign = cputime - cputimeBeforeFilterDesign;
151     end
152
153     if (jobOptions.boolTestFilter.if_fmimo)
154         filterIdx = filterIdx + 1;
155         filterSettings.if_fmimo.beta = thisBeta;
156         filterInfo.textKeys{filterIdx}  = 'if_fmimo';
157         filterInfo.fullNames{filterIdx} = 'Tikhonov IF: Full MIMO';
158         fprintf('  * %s\n',filterInfo.fullNames{filterIdx});
159         cputimeBeforeFilterDesign = cputime;
160         filterData.if_fmimo.inverseIRFsBBLS = ...
161             CreateInverseFilter(channelIRFsBBLS,'Tikhonov IF: Full MIMO',filterSettings.if_fmimo);
162         filterData.if_fmimo.receiverFilterBBLS = ones(1,size(channelIRFs,2));
163         filterData.if_fmimo.cputimeFilterDesign = cputime - cputimeBeforeFilterDesign;
164     end
165
166     %=========================================================================
167     % Convert the inverse filters and the receiver filters to the higher sampling rate
168     %=========================================================================
169     fprintf([' (b) Resample the inverse IRFs and the receiver from freqSamplingBB to ' ...
170              'freqSampling\n']);
171     if (config.downSampleForFilterDesign)
172         for filterIdx2 = 1:filterIdx;
173             fn = filterInfo.textKeys{filterIdx2};
```

```
174             filterData.(fn).inverseIRFsBB = ...
175                 ResampleIRFs(filterData.(fn).inverseIRFsBBLS,freqSampling,freqSamplingBB);
176             receiverFilterBBLSSize1 = size(filterData.(fn).receiverFilterBBLS,1);
177             % Don't bother re-sampling if it's a unity filter
178             if (receiverFilterBBLSSize1 == 1)
179                 filterData.(fn).receiverFilterBB = filterData.(fn).receiverFilterBBLS;
180             else
181                 filterData.(fn).receiverFilterBB = ...
182                     ResampleIRFs(filterData.(fn).receiverFilterBBLS,freqSampling,freqSamplingBB);
183             end
184         end
185     else
186         for filterIdx2 = 1:filterIdx;
187             fn = filterInfo.textKeys{filterIdx2};
188             filterData.(fn).inverseIRFsBB = filterData.(fn).inverseIRFsBBLS;
189         end
190     end
191
192     %=========================================================================
193     % Normalise the inverse and receiver filters
194     %=========================================================================
195     fprintf(' (c) Normalising the inverse filters according to root-mean-square:\n')
196     for filterIdx2 = 1:filterIdx;
197         fn = filterInfo.textKeys{filterIdx2};
198         inverseIRFsBBRMS = sum(abs(filterData.(fn).inverseIRFsBB).^2,1);
199         filterData.(fn).inverseIRFsBB = ...
200             filterData.(fn).inverseIRFsBB./sqrt(sum(inverseIRFsBBRMS(:)));
201
202         for ii = 1:size(channelIRFs,2)
203             receiverFilterBBRMS = sqrt(sum(abs(filterData.(fn).receiverFilterBB(:,ii)).^2));
204             filterData.(fn).receiverFilterBB(:,ii) = ...
205                 filterData.(fn).receiverFilterBB(:,ii)./receiverFilterBBRMS;
206         end
207     end
208
209     %=========================================================================
210     % TESTING STAGE
211     %=========================================================================
212     fprintf(' (d) Determining the full Virtual -> Reciever transfer function:\n')
213     for filterIdx2 = 1:filterIdx;
214         fn = filterInfo.textKeys{filterIdx2};
215         fprintf('  * %s \n', filterInfo.fullNames{filterIdx2});
216         filterData.(fn).fullResponseBB = ...
217             MatrixConvolve(channelIRFsBB,filterData.(fn).inverseIRFsBB);
218     end
219
220     fprintf(' (e) Iterative through each filter design\n');
221     modulationSignalBB = txSignalInfo.PSK.signalBBRaisedCosine(:,1);
222
223     for filterIdx2 = 1:filterIdx;
224         fn = filterInfo.textKeys{filterIdx2};
225         fprintf('  * %s \n', filterInfo.fullNames{filterIdx2});
226         for targetReceiverIdx1 = 1:length(config.targetReceiverRange)
227             targetReceiverIdx     = config.targetReceiverRange(targetReceiverIdx1);
228             crosstalkReceiverIdx = config.crosstalkReceiverRange(targetReceiverIdx1);
229             fprintf('   - Target receiver, crosstalk receiver: %d, %d\n', ...
230                     targetReceiverIdx,crosstalkReceiverIdx);
231
232             %=========================================================================
233             % Calculate transmission signal, and measure the peak and RMS of the signal
234             %=========================================================================
235             if (config.examineTransmitSignal)
236                 fprintf('     . Peak and power measurement of transmitter signals\n');
237                 matrixModulationSignalBB                          = zeros(size(modulationSignalBB,1),2,1);
238                 matrixModulationSignalBB(:,targetReceiverIdx) = modulationSignalBB;
```

```
239             filteredSignalForTxBB = ...
240                 MatrixConvolve(filterData.(fn).inverseIRFsBB,matrixModulationSignalBB);
241             measureTransmittedRMS = sqrt(mean(abs(filteredSignalForTxBB(:).^2)));
242             measureTransmittedPeak  = max(abs(filteredSignalForTxBB(:)));
243             thisResults.(fn).txpeak(betaIdx,targetReceiverIdx)  = measureTransmittedPeak;
244             thisResults.(fn).txpower(betaIdx,targetReceiverIdx) = measureTransmittedRMS;
245         end
246
247         %=========================================================================
248         % Calculate the target receiver signal, and measure various powers
249         %=========================================================================
250         % The receiver signal power is measured for three regions of the received signal:
251         %
252         %  [.........SSSSSSSSSSSSSSSSSS..]
253         %  |<----------------0---------------->|
254         %  |<---1--->|                       |
255         %   |        |<----------2---------->|
256         %            |
257         %            peakIndex
258         %
259         % where peakIndex is the index of the peak in the full response, and should also be
260         % the delay of the signal.
261         % Essentially:
262         %     0 the entire received power
263         %     1 the ISI noise floor
264         %     2 the data signal
265         %
266         fprintf(['    . Calculate the signal at the target receiver' ...
267                  '\n']);
268         targetReceiverSignalBB = conv( ...
269             filterData.(fn).fullResponseBB(:,targetReceiverIdx,targetReceiverIdx), ...
270             modulationSignalBB ...
271         );
272         [dummy peakIndex] = ...
273             max(filterData.(fn).fullResponseBB(:,targetReceiverIdx,targetReceiverIdx));
274         measureReceiverSignalRMS0      = sqrt(mean(abs( ...
275             targetReceiverSignalBB).^2));
276         measureReceiverSignalRMS0Time = length(targetReceiverSignalBB) * freqSampling;
277         measureReceiverSignalRMS1      = sqrt(mean(abs( ...
278             targetReceiverSignalBB(1:peakIndex)).^2));
279         measureReceiverSignalRMS1Time = peakIndex * freqSampling;
280         measureReceiverSignalRMS2      = sqrt(mean(abs( ...
281             targetReceiverSignalBB(peakIndex + 1 ...
282                                    :length(txSignalInfo.PSK.complexSymbols)*symbolLength)).^2));
283         measureReceiverSignalRMS2Time = ...
284             length(txSignalInfo.PSK.complexSymbols) * symbolLength * freqSampling;
285
286         thisResults.(fn).prerx_power(betaIdx,targetReceiverIdx)    = ...
287             measureReceiverSignalRMS0;
288         thisResults.(fn).prerx_power_t(betaIdx,targetReceiverIdx)  = ...
289             measureReceiverSignalRMS0Time;
290         thisResults.(fn).prerx_power1(betaIdx,targetReceiverIdx)   = ...
291             measureReceiverSignalRMS1;
292         thisResults.(fn).prerx_power1_t(betaIdx,targetReceiverIdx) = ...
293             measureReceiverSignalRMS1Time;
294         thisResults.(fn).prerx_power2(betaIdx,targetReceiverIdx)   = ...
295             measureReceiverSignalRMS2;
296         thisResults.(fn).prerx_power2_t(betaIdx,targetReceiverIdx) = ...
297             measureReceiverSignalRMS2Time;
298
299         %=========================================================================
300         % Apply the receiver filter, and measure the total power
301         %=========================================================================
302         fprintf('    . Apply the receiver filter\n');
303         targetReceiverSignalRXFilteredBB = conv( ...
```

```
304                 targetReceiverSignalBB, ...
305                 filterData.(fn).receiverFilterBB(:,targetReceiverIdx));
306
307             measureReceiverSignalRXFilteredRMS0      = ...
308                 sqrt(mean(abs(targetReceiverSignalRXFilteredBB).^2));
309             measureReceiverSignalRXFilteredRMS0Time = ...
310                 length(targetReceiverSignalRXFilteredBB) * freqSampling;
311
312             thisResults.(fn).postrx_power(betaIdx,targetReceiverIdx)    = ...
313                 measureReceiverSignalRXFilteredRMS0;
314             thisResults.(fn).postrx_power_t(betaIdx,targetReceiverIdx) = ...
315                 measureReceiverSignalRXFilteredRMS0Time;
316
317             %=========================================================================
318             % Perform symbol synchronisation
319             %=========================================================================
320             % Use the function, FindSignalFirstPeak to find the first peak in the signal.
321             % If the returned firstPeakIndex is empty, or does not allow enough samples to
322             % obtain the transmitted signal, then signal is obviously too distorted to obtain any
323             % samples, and thus further processing should be skipped.
324             %
325             fprintf('     . Performing symbol synchronisation\n');
326             signalLength = length(targetReceiverSignalRXFilteredBB);
327             % We need this many points after the firstPeakIndex, so don't bother searching for a
328             % peak in them.
329             samplesNeeded = symbolLength*length(txSignalInfo.PSK.complexSymbols);
330             [firstPeakIndex firstPeakAmpPhaseInverse] = ...
331                 FindSignalFirstPeak( ...
332                     targetReceiverSignalRXFilteredBB(1:signalLength-samplesNeeded), ...
333                     symbolLength);
334
335             performSymbolDetection = 1;
336             if length(firstPeakIndex) == 0
337                 performSymbolDetection = 0;
338             else
339                 if (firstPeakIndex + symbolLength + length(txSignalInfo.PSK.complexSymbols) - 1) ...
340                         > length(targetReceiverSignalRXFilteredBB)
341                     performSymbolDetection = 0;
342                 end
343             end
344             if performSymbolDetection == 0
345                 disp('Skipping!');
346                 thisResults.(fn).skipping(betaIdx,targetReceiverIdx) = 1;
347             else
348                 thisResults.(fn).skipping(betaIdx,targetReceiverIdx) = 0;
349
350                 % Store the first peak index
351                 thisResults.(fn).idx(betaIdx,targetReceiverIdx) = firstPeakIndex;
352
353                 %=========================================================================
354                 % Sample the signal at the sampling instances (normal and fractional)
355                 %=========================================================================
356                 sampledSignal     = ...
357                     targetReceiverSignalRXFilteredBB( ...
358                         firstPeakIndex + ...
359                         symbolLength*[ ...
360                             txSignalInfo.PSK.leadInSequenceLength ...
361                             :length(txSignalInfo.PSK.complexSymbols) - 1]).';
362
363                 sampledSignalFS = ...
364                     targetReceiverSignalRXFilteredBB( ...
365                         firstPeakIndex + ...
366                         floor([ ...
367                             symbolLength*txSignalInfo.PSK.leadInSequenceLength ...
368                             :symbolLength/config.fractionalSpacing ...
```

```
369                 :symbolLength*length(txSignalInfo.PSK.complexSymbols) - 1])).';
370
371             %=========================================================================
372             % Compute some standard deviations
373             %=========================================================================
374             % The standard deviations are calculated for two cases, both against the original
375             % transmitted complex sequence, and adjusting the phase and amplitude according to
376             % the mean difference.  The two methods of measuring the standard deviation are:
377             %
378             % 1. Isolating the various constellation positions, and computing the standard
379             %    deviation fo each position, and performing the appropriate summation.
380             %
381             % 2. Subtracting the original complex sequence from the phase and amplitude
382             %    compensated signal and calculating the standard deviation of this signal.
383             %
384             % Note that both methods should be effectively the same.
385             %
386             fprintf('    . Computing the standard deviations\n');
387
388             % Obtain the original complex sequence
389             originalComplexSequence = ...
390                 txSignalInfo.PSK.complexSymbols( ...
391                     txSignalInfo.PSK.leadInSequenceLength+1 ...
392                     :length(txSignalInfo.PSK.complexSymbols));
393
394             pskIndexesPos1 = find(real(originalComplexSequence) > 0.9);
395             pskIndexesPosI = find(imag(originalComplexSequence) > 0.9);
396             pskIndexesNeg1 = find(real(originalComplexSequence) < -0.9);
397             pskIndexesNegI = find(imag(originalComplexSequence) < -0.9);
398
399             % Number of samples at [1 i -1 -i] respectively
400             pskIndexesCounts = [ ...
401                 length(pskIndexesPos1) ...
402                 length(pskIndexesPosI) ...
403                 length(pskIndexesNeg1) ...
404                 length(pskIndexesNegI) ...
405                                 ];
406
407             % Average amplitude / phase adjustment for all symbols.
408             meanSymbolAmpPhaseInverse = 1./(mean( ...
409                 sampledSignal([ ...
410                     pskIndexesPos1 ...
411                     pskIndexesPosI ...
412                     pskIndexesNeg1 ...
413                     pskIndexesNegI]) ...
414                 ./ ...
415                 originalComplexSequence([ ...
416                     pskIndexesPos1 ...
417                     pskIndexesPosI ...
418                     pskIndexesNeg1 ...
419                     pskIndexesNegI]) ...
420                 ));
421
422             % Standard deviation each of the various constellation position after applying
423             % the amplitude and phase adjustment. The standard deviation is calculated
424             % against the meanSymbolAmpPhaseInverse, so the gain is slightly different than
425             % firstPeakAmpPhaseInverse!
426
427             pskIndexesStandardDeviationsAPAdjusted = [ ...
428                 std(meanSymbolAmpPhaseInverse * sampledSignal(pskIndexesPos1)) ...
429                 std(meanSymbolAmpPhaseInverse * sampledSignal(pskIndexesPosI)) ...
430                 std(meanSymbolAmpPhaseInverse * sampledSignal(pskIndexesNeg1)) ...
431                 std(meanSymbolAmpPhaseInverse * sampledSignal(pskIndexesNegI))];
432
433             % Calculate standard deviation from the summation of each individual
```

```
434                 % constellation position.
435                 measureStandardDeviation1 = ...
436                     sqrt(sum( ...
437                         pskIndexesStandardDeviationsAPAdjusted.^2 .* pskIndexesCounts ...
438                         )/sum(pskIndexesCounts));
439
440                 % Calculate standard deviation from the difference between amplitude and phase
441                 % adjusted sampled sequence and the original complex sequence.
442                 measureStandardDeviation2 = ...
443                     std(meanSymbolAmpPhaseInverse * sampledSignal - originalComplexSequence);
444
445                 % Store the mean symbol amplitude and phase compensation
446                 thisResults.(fn).gain(betaIdx,targetReceiverIdx) = abs(meanSymbolAmpPhaseInverse);
447
448                 % Store the measured standard deviations.
449                 thisResults.(fn).total_std(betaIdx,targetReceiverIdx)  = measureStandardDeviation1;
450                 thisResults.(fn).total_std2(betaIdx,targetReceiverIdx) = measureStandardDeviation2;
451
452                 %=========================================================================
453                 % Compute the symbol error
454                 %=========================================================================
455                 % The symbol error is computed by passing both the received signal, and the original
456                 % transmitted sequence through the detector and counting all the instances where
457                 % they are not the same.
458                 %
459                 fprintf('     . Computing the symbol error\n');
460                 basisInfo.dummyVariable = 1;
461
462                 [receivedSignalSignalError receivedSignalDetectedSymbols] = ...
463                     DetectorNonAdaptive( ...
464                         'PSK', ...
465                         [ ...
466                             real(meanSymbolAmpPhaseInverse*sampledSignal); ...
467                             imag(meanSymbolAmpPhaseInverse*sampledSignal)].', ...
468                         basisInfo,txSignalInfo.PSK.modulationSettings);
469
470                 [originalComplexSequenceSignalError originalComplexSequenceDetectedSymbols] = ...
471                     DetectorNonAdaptive( ...
472                         'PSK', ...
473                         [ ...
474                             real(originalComplexSequence) ; ...
475                             imag(originalComplexSequence) ...
476                         ].', ...
477                         basisInfo,txSignalInfo.PSK.modulationSettings);
478
479                 measureSymbolError = sum([ ...
480                     receivedSignalDetectedSymbols~=originalComplexSequenceDetectedSymbols ...
481                   ] ~= 0);
482
483                 % Store the measured symbol error
484                 thisResults.(fn).symbolerr(betaIdx,targetReceiverIdx) = measureSymbolError;
485
486                 %=========================================================================
487                 % Run the adaptive filter algorithms
488                 %=========================================================================
489                 if (config.performAdaptiveFiltering)
490                     fprintf('     . Running the adaptive filtering algorithms\n');
491                     %-------------------------------------------------------------------------
492                     % Initialise some parameters for the filters
493                     %-------------------------------------------------------------------------
494                     constellationValues = ...
495                         GetConstellationValues('PSK',0,txSignalInfo.PSK.modulationSettings);
496
497                     zfa.filterTaps   = [1 zeros(1,config.adaptiveFilterLength)];
498                     zfat.filterTaps  = [1 zeros(1,config.adaptiveFilterLength)];
```

```
499 msea.filterTaps  = ...
500     [zeros(1,config.filterPeakIndex-1) ...
501      1 ...
502      zeros(1,config.adaptiveFilterLength-config.filterPeakIndex)];
503 rlsa.filterTaps  = ...
504     [zeros(1,config.filterPeakIndex-1) ...
505      1 ...
506      zeros(1,config.adaptiveRLSFilterLength-config.filterPeakIndex)];
507 mseat.filterTaps = msea.filterTaps;
508 rlsat.filterTaps = rlsa.filterTaps;
509
510 msea.feedbackFilterTaps  = zeros(1,config.adaptiveFeedbackFilterLength);
511 rlsa.feedbackFilterTaps  = zeros(1,config.adaptiveRLSFeedbackFilterLength);
512 mseat.feedbackFilterTaps = msea.feedbackFilterTaps;
513 rlsat.feedbackFilterTaps = rlsa.feedbackFilterTaps;
514
515 %-------------------------------------------------------------------------
516 % Initialise some history collection parameters
517 %-------------------------------------------------------------------------
518 zfa.averageErrorHistory   = [];
519 zfat.averageErrorHistory  = [];
520 msea.averageErrorHistory  = [];
521 mseat.averageErrorHistory = [];
522 rlsa.averageErrorHistory  = [];
523 rlsat.averageErrorHistory = [];
524
525 zfa.signalErrorHistory   = [];
526 zfat.signalErrorHistory  = [];
527 msea.signalErrorHistory  = [];
528 mseat.signalErrorHistory = [];
529
530 %-------------------------------------------------------------------------
531 % Define two training sequences (none, and 40 symbols)
532 %-------------------------------------------------------------------------
533 trainingSequenceNone       = [];
534 trainingSequence40Symbols = txSignalInfo.PSK.complexSymbols( ...
535     txSignalInfo.PSK.leadInSequenceLength + [1:40] ...
536     );
537
538 %-------------------------------------------------------------------------
539 % Run the RLSA algorithm (with and without a training sequence) once
540 %-------------------------------------------------------------------------
541 if (config.runRLSAAlgorithm)
542     % Two passes of RLS are performed to allow the rlsa algorithm to filter
543     % the first few samples better than it would with just one pass
544     for ii = 1:2
545         % Run the algorithm for no training sequence.
546         [rlsa.signalError rlsa.detectedSymbols rlsa.filterTaps ...
547          rlsa.filterTapHistory rlsa.filteredSignal rlsa.feedbackFilterTaps] ...
548             = DetectorAdaptiveRLS( ...
549                 constellationValues, ...
550                 meanSymbolAmpPhaseInverse*sampledSignalFS.', ...
551                 0.999, ...
552                 rlsa.filterTaps, ...
553                 trainingSequenceNone, ...
554                 config.fractionalSpacing, ...
555                 rlsa.feedbackFilterTaps, ...
556                 config.filterPeakIndex);
557
558         % Run the algorithm with a training sequence of 40 Symbols
559         [rlsat.signalError rlsat.detectedSymbols rlsat.filterTaps ...
560          rlsat.filterTapHistory rlsat.filteredSignal ...
561          rlsat.feedbackFilterTaps] ...
562             = DetectorAdaptiveRLS( ...
563                 constellationValues, ...
```

```
564                         meanSymbolAmpPhaseInverse*sampledSignalFS.', ...
565                         0.999, ...
566                         rlsat.filterTaps, ...
567                         trainingSequence40Symbols, ...
568                         config.fractionalSpacing, ...
569                         rlsat.feedbackFilterTaps, ...
570                         config.filterPeakIndex);
571
572                     % Cumulate the average error history
573                     rlsa.averageErrorHistory ...
574                         = [rlsa.averageErrorHistory mean(abs(rlsa.signalError))];
575                     rlsat.averageErrorHistory ...
576                         = [rlsat.averageErrorHistory mean(abs(rlsat.signalError))];
577                     fprintf('          RLSA: %.4f  RLSA(T): %.4f\n', ...
578                         mean(abs(rlsa.signalError)), ...
579                         mean(abs(rlsat.signalError)));
580                 end
581             else
582                 fprintf('    Skipping rlsa\n');
583             end
584
585             %=========================================================================
586             % Loop through all the deltaSteps, and perform adaptive filtering
587             %=========================================================================
588             for deltaStepIdx = 1:length(config.deltaSteps)
589                 thisDelta = config.deltaSteps(deltaStepIdx);
590                 fprintf('      thisDelta %.4f::',thisDelta);
591
592                 % Run the zero forcing algorithm without a training sequence
593                 [zfa.signalError        zfa.detectedSymbols    zfa.filterTaps ...
594                  zfa.filterTapHistory   zfa.filteredSignal] ...
595                     = DetectorAdaptiveZF(constellationValues, ...
596                                          meanSymbolAmpPhaseInverse*sampledSignal.', ...
597                                          thisDelta, ...
598                                          zfa.filterTaps, ...
599                                          trainingSequenceNone);
600
601                 % Run the LMS algorithm without a training sequence
602                 [msea.signalError       msea.detectedSymbols   msea.filterTaps ...
603                  msea.filterTapHistory  msea.filteredSignal    msea.feedbackFilterTaps] ...
604                     = DetectorAdaptiveMSE(constellationValues, ...
605                                           meanSymbolAmpPhaseInverse*sampledSignal.', ...
606                                           thisDelta, ...
607                                           msea.filterTaps, ...
608                                           trainingSequenceNone, ...
609                                           1, ...
610                                           msea.feedbackFilterTaps, ...
611                                           config.filterPeakIndex);
612
613                 % Run the zero forcing algorithm with a training sequence of 40 symbols
614                 [zfat.signalError       zfat.detectedSymbols   zfat.filterTaps ...
615                  zfat.filterTapHistory  zfat.filteredSignal] ...
616                     = DetectorAdaptiveZF(constellationValues, ...
617                                          meanSymbolAmpPhaseInverse*sampledSignal.', ...
618                                          thisDelta, ...
619                                          zfat.filterTaps, ...
620                                          trainingSequence40Symbols);
621
622                 % Run the LMS algorithm with a training sequence of 40 symbols
623                 [mseat.signalError      mseat.detectedSymbols  mseat.filterTaps ...
624                  mseat.filterTapHistory mseat.filteredSignal   mseat.feedbackFilterTaps] ...
625                     = DetectorAdaptiveMSE(constellationValues, ...
626                                           meanSymbolAmpPhaseInverse*sampledSignal.', ...
627                                           thisDelta, ...
628                                           mseat.filterTaps, ...
```

```
                                    trainingSequence40Symbols, ...
                                    1, ...
                                    mseat.feedbackFilterTaps, ...
                                    config.filterPeakIndex);

    fprintf(' ZFA: %.4f  ZFA(T): %.4f MSEA: %.4f  MSEA(T): %.4f\n', ...
            mean(abs(zfa.signalError)) ,mean(abs(zfat.signalError)), ...
            mean(abs(msea.signalError)),mean(abs(mseat.signalError)));

    %-----------------------------------------------------------------------
    % Reset filter taps if error increase since last deltaStep
    %-----------------------------------------------------------------------
    if (deltaStepIdx == 1)
        lastDelta.zfa.averageSignalError   = mean(abs(zfa.signalError));
        lastDelta.zfat.averageSignalError  = mean(abs(zfat.signalError));
        lastDelta.msea.averageSignalError  = mean(abs(msea.signalError));
        lastDelta.mseat.averageSignalError = mean(abs(mseat.signalError));
    else
        if ( lastDelta.zfa.averageSignalError > mean(abs(zfa.signalError)) )
            lastDelta.zfa.averageSignalError   = mean(abs(zfa.signalError));
        else
            disp('zfa taps reset!');
            zfa.filterTaps = lastDelta.zfa.filterTaps;
        end

        if ( lastDelta.zfat.averageSignalError > mean(abs(zfat.signalError)) )
            lastDelta.zfat.averageSignalError = mean(abs(zfat.signalError));
        else
            disp('zfat taps reset!');
            zfat.filterTaps = lastDelta.zfat.filterTaps;
        end

        if ( lastDelta.msea.averageSignalError > mean(abs(msea.signalError)) )
            lastDelta.msea.averageSignalError = mean(abs(msea.signalError));
        else
            disp('msea taps reset!');
            msea.filterTaps = lastDelta.msea.filterTaps;
            msea.feedbackFilterTaps = lastDelta.msea.feedbackFilterTaps;
        end

        if ( lastDelta.mseat.averageSignalError > mean(abs(mseat.signalError)) )
            lastDelta.mseat.averageSignalError = mean(abs(mseat.signalError));
        else
            disp('mseat taps reset!');
            mseat.filterTaps = lastDelta.mseat.filterTaps;
            mseat.feedbackFilterTaps = lastDelta.mseat.feedbackFilterTaps;
        end
    end

    lastDelta.zfa.filterTaps   = zfa.filterTaps;
    lastDelta.zfat.filterTaps  = zfat.filterTaps;
    lastDelta.msea.filterTaps  = msea.filterTaps;
    lastDelta.mseat.filterTaps = mseat.filterTaps;
    lastDelta.msea.feedbackFilterTaps  = msea.feedbackFilterTaps;
    lastDelta.mseat.feedbackFilterTaps = mseat.feedbackFilterTaps;

    % Cumulate the signal error history
    zfa.signalErrorHistory   = [zfa.signalErrorHistory   zfa.signalError];
    zfat.signalErrorHistory  = [zfat.signalErrorHistory  zfat.signalError];
    msea.signalErrorHistory  = [msea.signalErrorHistory  msea.signalError];
    mseat.signalErrorHistory = [mseat.signalErrorHistory mseat.signalError];

    % Cumulate the average error history
    zfa.averageErrorHistory   = ...
        [zfa.averageErrorHistory   mean(abs(zfa.signalError))];
```

```
694                 msea.averageErrorHistory  = ...
695                     [msea.averageErrorHistory  mean(abs(msea.signalError))];
696                 zfat.averageErrorHistory  = ...
697                     [zfat.averageErrorHistory  mean(abs(zfat.signalError))];
698                 mseat.averageErrorHistory = ...
699                     [mseat.averageErrorHistory mean(abs(mseat.signalError))];
700             end
701
702             %=========================================================================
703             % Record the average error history
704             %=========================================================================
705             if (config.runRLSAAlgorithm)
706                 thisResults.(fn).rle_ee(:,betaIdx,targetReceiverIdx)  = ...
707                     rlsa.averageErrorHistory;
708                 thisResults.(fn).rlet_ee(:,betaIdx,targetReceiverIdx) = ...
709                     rlsat.averageErrorHistory;
710             end
711             thisResults.(fn).zfa_ee(:,betaIdx,targetReceiverIdx)   =  ...
712                 zfa.averageErrorHistory;
713             thisResults.(fn).zfat_ee(:,betaIdx,targetReceiverIdx)  =  ...
714                 zfat.averageErrorHistory;
715             thisResults.(fn).msea_ee(:,betaIdx,targetReceiverIdx)  = ...
716                 msea.averageErrorHistory;
717             thisResults.(fn).mseat_ee(:,betaIdx,targetReceiverIdx) = ...
718                 mseat.averageErrorHistory;
719
720             %=========================================================================
721             % Record the standard deviation of the adaptive filtered signal
722             %=========================================================================
723             if (config.runRLSAAlgorithm)
724                 rlsa.filteredSignalErrors  = rlsa.filteredSignal  - originalComplexSequence;
725                 rlsat.filteredSignalErrors = rlsat.filteredSignal - originalComplexSequence;
726             end
727             zfa.filteredSignalErrors   = zfa.filteredSignal   - originalComplexSequence;
728             msea.filteredSignalErrors  = msea.filteredSignal  - originalComplexSequence;
729             zfat.filteredSignalErrors  = zfat.filteredSignal  - originalComplexSequence;
730             mseat.filteredSignalErrors = mseat.filteredSignal - originalComplexSequence;
731
732             if (config.runRLSAAlgorithm)
733                 rlsa.filteredSignalErrorStdDev  = std(rlsa.filteredSignalErrors);
734                 rlsat.filteredSignalErrorStdDev = std(rlsa.filteredSignalErrors);
735             end
736             zfa.filteredSignalErrorStdDev   = std(zfa.filteredSignalErrors);
737             zfat.filteredSignalErrorStdDev  = std(zfat.filteredSignalErrors);
738             msea.filteredSignalErrorStdDev  = std(msea.filteredSignalErrors);
739             mseat.filteredSignalErrorStdDev = std(mseat.filteredSignalErrors);
740
741             if (config.runRLSAAlgorithm)
742                 thisResults.(fn).rlsa_std(betaIdx,targetReceiverIdx)  = ...
743                     rlsa.filteredSignalErrorStdDev;
744                 thisResults.(fn).rlsat_std(betaIdx,targetReceiverIdx) = ...
745                     rlsat.filteredSignalErrorStdDev;
746             end
747             thisResults.(fn).zfa_std(betaIdx,targetReceiverIdx)   = ...
748                 zfa.filteredSignalErrorStdDev;
749             thisResults.(fn).zfat_std(betaIdx,targetReceiverIdx)  = ...
750                 zfat.filteredSignalErrorStdDev;
751             thisResults.(fn).msea_std(betaIdx,targetReceiverIdx)  = ...
752                 msea.filteredSignalErrorStdDev;
753             thisResults.(fn).mseat_std(betaIdx,targetReceiverIdx) = ...
754                 mseat.filteredSignalErrorStdDev;
755
756             %=========================================================================
757             % Record the symbol error of the adaptive filtered signal
758             %=========================================================================
```

```
759                 if (config.runRLSAAlgorithm)
760                     rlsa.detectedSymbolsErrorCount    = ...
761                         sum(rlsa.detectedSymbols  - originalComplexSequence ~= 0);
762                     rlsat.detectedSymbolsErrorCount  = ...
763                         sum(rlsat.detectedSymbols - originalComplexSequence ~= 0);
764                 end
765                 zfa.detectedSymbolsErrorCount     = ...
766                     sum(zfa.detectedSymbols    - originalComplexSequence ~= 0);
767                 zfat.detectedSymbolsErrorCount    = ...
768                     sum(zfat.detectedSymbols  - originalComplexSequence ~= 0);
769                 msea.detectedSymbolsErrorCount    = ...
770                     sum(msea.detectedSymbols  - originalComplexSequence ~= 0);
771                 mseat.detectedSymbolsErrorCount  = ...
772                     sum(mseat.detectedSymbols - originalComplexSequence ~= 0);
773
774                 if (config.runRLSAAlgorithm)
775                     thisResults.(fn).rlsa_symbolerr(betaIdx,targetReceiverIdx)  = ...
776                         rlsa.detectedSymbolsErrorCount;
777                     thisResults.(fn).rlsat_symbolerr(betaIdx,targetReceiverIdx) = ...
778                         rlsat.detectedSymbolsErrorCount;
779                 end
780                 thisResults.(fn).zfa_symbolerr(betaIdx,targetReceiverIdx)    = ...
781                     zfa.detectedSymbolsErrorCount;
782                 thisResults.(fn).zfat_symbolerr(betaIdx,targetReceiverIdx)  = ...
783                     zfat.detectedSymbolsErrorCount;
784                 thisResults.(fn).msea_symbolerr(betaIdx,targetReceiverIdx)  = ...
785                     msea.detectedSymbolsErrorCount;
786                 thisResults.(fn).mseat_symbolerr(betaIdx,targetReceiverIdx) = ...
787                     mseat.detectedSymbolsErrorCount;
788             end
789
790             %=========================================================================
791             % Examine the cross-talk
792             %=========================================================================
793             if (config.examineCrossTalk)
794                 fprintf('        . Examining the cross talk\n');
795                 % Determine the index range of the received signal at which a signal
796                 % would received desired to be transmitted to the cross talk location having the
797                 % same duration as the transmitted signal for (1) before and (2) after the
798                 % receiver filter.
799                 %
800                 [dummy crosstalkPeakIndex1] = max( ...
801                     filterData.(fn).fullResponseBB(:,crosstalkReceiverIdx,crosstalkReceiverIdx));
802                 crossTalkSignalRangePreRXFilter = ...
803                     crosstalkPeakIndex1 ...
804                     + [1:symbolLength * (length(txSignalInfo.PSK.complexSymbols) - 1)];
805
806                 [dummy crosstalkPeakIndex2] = max(conv( ...
807                     filterData.(fn).fullResponseBB(:,crosstalkReceiverIdx,crosstalkReceiverIdx), ...
808                     filterData.(fn).receiverFilterBBLS(:,crosstalkReceiverIdx)));
809                 crossTalkSignalRangePostRXFilter  = ...
810                     crosstalkPeakIndex2 ...
811                     + [1:symbolLength * (length(txSignalInfo.PSK.complexSymbols) - 1)];
812
813                 % Calculate the entire cross-talk on the target Receiver from a signal
814                 % desired for the cross-talk receiver.
815                 crosstalkReceiverSignal = conv( ...
816                     filterData.(fn).fullResponseBB(:,targetReceiverIdx,crosstalkReceiverIdx), ...
817                     modulationSignalBB);
818
819                 % Apply the receiver filter to the received signal
820                 crosstalkReceiverSignalPostRXFilter = conv( ...
821                     crosstalkReceiverSignal, ...
822                     filterData.(fn).receiverFilterBB(:,targetReceiverIdx));
823
```

```
824                     measureCrosstalkPreRXSignalRMS       = sqrt(mean(abs(crosstalkReceiverSignal).^2));
825                     measureCrosstalkPreRXSignalRMSTime  = ...
826                         sqrt(mean(abs(crosstalkReceiverSignal(crossTalkSignalRangePreRXFilter)).^2));
827                     measureCrosstalkPostRXSignalRMS      = ...
828                         sqrt(mean(abs(crosstalkReceiverSignalPostRXFilter).^2));
829                     measureCrosstalkPostRXSignalRMSTime = ...
830                         sqrt(mean(abs( ...
831                             crosstalkReceiverSignalPostRXFilter(crossTalkSignalRangePostRXFilter)).^2));
832
833                     % Record measurements
834                     thisResults.(fn).crosstalk_prerx(betaIdx,targetReceiverIdx)       = ...
835                         measureCrosstalkPreRXSignalRMS;
836                     thisResults.(fn).crosstalk_postrx(betaIdx,targetReceiverIdx)      = ...
837                         measureCrosstalkPostRXSignalRMS;
838                     thisResults.(fn).crosstalk_postrx_rng(betaIdx,targetReceiverIdx) = ...
839                         measureCrosstalkPostRXSignalRMSTime;
840                 end
841             end
842             thisResults.(fn).cputime(betaIdx) = filterData.(fn).cputimeFilterDesign;
843         end
844     end
845     thisResults.times(betaIdx) = toc - tocStartBetaLoop;
846     tocStartBetaLoop = toc;
847 end
848 thisResults.totaltime = toc;
```

A.4 Thesis MATLAB library

The following MATLAB scripts form a library of functions used in the scripts required to perform both Experiments 1 and 2.

A.4.1 BasebandToPassband.m

```
1  function outSignal = BasebandToPassband(inSignal,freqCarrier,freqSampling)
2  % USAGE: outSignal = BasebandToPassband(inSignal,freqCarrier,freqSampling)
3  %
4  % This function converts the three dimensinal complex signals, inSignal, from baseband to Passband
5  % using formula (4.1-14) given in [Proakis 2001], Section 4.1.1, Page 151:
6  %
7  %       s(t) = Re[s(l)e^{j2\pif_{c}t}
8  %
9
10 sampleLength = size(inSignal,1);
11 outSignal = zeros(size(inSignal));
12 for jj = 1:size(inSignal,2)
13     for kk = 1:size(inSignal,3)
14         outSignal(:,jj,kk) = real(inSignal(:,jj,kk) ...
15                                   .*exp(j*2*pi*(freqCarrier/freqSampling)*[0:(sampleLength-1)]') ...
16                                   )';
17     end
18 end
```

A.4.2 BitSequenceToBlockValues.m

```
function outNumericalValues = BitSequenceToBlockValues(inSequence,blockSize)
% USAGE: outNumericalValues = BitSequenceToBlockValues(inSequence,blockSize)
%
% This function takes the input stream, inSequence, which should consist of binary values (0 or 1)
% and converts it into a sequence of numerical values, outNumericalValues, made from blocks of
% blockSize bits.

% Convert into a row vector (in case it wasn't)
inSequence2 = inSequence(:)';

% Append zeros so that we can resize it easily
if (rem(size(inSequence2,2),blockSize) > 0)
    inSequence2 = [inSequence2 zeros(1, blockSize-rem(size(inSequence2,2),blockSize))];
end

sequenceBlockMatrix = reshape(inSequence2,blockSize,fix(size(inSequence2,2)/blockSize));
outNumericalValues  = (2.^[blockSize-1:-1:0])*sequenceBlockMatrix;
```

A.4.3 BitSequenceToComplexSequence.m

```
function outComplexSequence = ...
    BitSequenceToComplexSequence(inSequence,modulationMethod,modulationsSettings)
% USAGE: outComplexSequence = ...
%             BitSequenceToComplexSequence(inSequence,modulationMethod,modulationsSettings)
%
% This function takes the input stream, inSequence, which should consist of binary values (0 or 1)
% and converts it into a sequence of complex values according modulation method given by the
% variable, modulationMethod. Extra settings for the modulation method can be provided through
% the variable, modulationsSettings.

if strcmp(modulationMethod,'DSB_PAM')
    % Implementation of Pulse Amplitude Modulation (PAM)
    % See [Proakis 2001] - Page 169, 4.3.1
    k = modulationsSettings.k;

    % Split input stream into blocks of k bits
    inSequenceBlocked = BitSequenceToBlockValues(inSequence,k);

    % Convert each block into the grey encoded value:
    greyMap = GreyEncodeMap(k);

    % Normalise so the values are between -1 and 1.
    greyMap = (greyMap*2 - 2^k + 1)./(2^k-1);

    % Map the block values to complex values
    outComplexSequence = greyMap(inSequenceBlocked+1);

elseif strcmp(modulationMethod,'PSK')
    % Implementation of Phase Shift Keying
    % See [Proakis 2001] - Page 171, 4.3.1
    k = modulationsSettings.k;

    % Split input stream into blocks of k bits
    inSequenceBlocked = BitSequenceToBlockValues(inSequence,k);

    % Convert each block into the grey encoded value:
    greyMap = GreyEncodeMap(k);

    % Map the block values to complex values
    inSequenceBlockedMapped = greyMap(inSequenceBlocked+1);

```

```
42      % Create the complex vector
43      outComplexSequence = exp(j*2*pi*(inSequenceBlockedMapped-1)/(2^k) + j*modulationsSettings.addphase);
44
45  elseif strcmp(modulationMethod, 'QAM')
46      % Implementation of Quadrature Amplitude Modulation
47      % See [Proakis 2001] - Page 174, 4.3.1,
48      % This method requires that a constellation map is provided.
49
50      k = modulationsSettings.k;
51
52      % Split input stream into blocks of k bits
53      inSequenceBlocked = BitSequenceToBlockValues(inSequence,k);
54
55      % Map the block values to complex values
56      outComplexSequence = modulationsSettings.constellation(inSequenceBlocked+1).';
57  end
```

A.4.4 CentralPeakSignalTrim.m

```
1   function outSignal = CentralPeakSignalTrim(inSignal,trimdBLevel,blockSize)
2   % USAGE: outSignal = CentralPeakSignalTrim(inSignal,trimdBLevel,blockSize)
3   %
4   % This function performs a trim about the peak value to the times at which the signals to less than
5   % timdBLevel dB, rounding up to integer values of blockSize.
6
7   [maxValue peakIndex] = max(abs(inSignal));
8
9   % ignore the log10(0) warnings.
10  warning off
11  minIndex = min(find(20*log10(abs(inSignal) / maxValue) > trimdBLevel));
12  maxIndex = max(find(20*log10(abs(inSignal) / maxValue) > trimdBLevel));
13  warning on
14
15  halfTrimRange = max([peakIndex - minIndex maxIndex - peakIndex]);
16  halfTrimRange = ceil(halfTrimRange/blockSize)*blockSize;
17
18  outSignal = inSignal(peakIndex + [-halfTrimRange:halfTrimRange-1]);
```

A.4.5 ComplexSequenceToSignal.m

```
1   function outSignalBB = ComplexSequenceToSignal(inComplexSequence,spectralShapingFilter,symbolLength)
2   % USAGE: outSignalBB = ComplexSequenceToSignal(inComplexSequence,spectralShapingFilter,symbolLength)
3   %
4   % This function takes the input sequence, inComplexSequence, which should consist of complex values
5   % and using the spectral shaping filter, spectralShapingFilter, creates a base-band signal having
6   % symbols of symbolLength long. The spectral shaping filter must have been created for the specific
7   % symbolLength provided, and be an integer number symbol lengths.
8   %
9   % The output signal is formed from overlapping the signals in the matrix, i.e. the output should
10  % look like:
11  %
12  %  outputSignalBB = [ms(1:sfl,1)                zeros(...)]
13  %                 + [zeros(1:1*sl) ms(1:sfl,2) zeros(...)]
14  %                 + [zeros(1:2*sl) ms(1:sfl,3) zeros(...)]
15  %
16  % where
17  %    ms  = matrixSymbolsShapedToSignals
18  %    sfl = length(spectralShapingFilter)
19  %    syl = symbolLength
20  %    ... is the zero-padding to the end of the matrix... (dependent on number of symbols)
```

```
%
% However, since this requires a for loop to loop through every symbol, MATLAB can be rather slow at
% this process. To reduce the for-loop size, that calculations can equivalently implemented
% as:
%
% outputSignalBB = [ms([1:syl]+0,1) ms([1:syl]+0,2) ms([1:syl]+0,3) ms([1:syl]+0,4)...]
%                  + [0....,.......0 ms([1:syl]+1,1) ms([1:syl]+1,2) ms([1:syl]+1,3)...]
%                  + [0.....,,.....................0 ms([1:syl]+2,1) ms([1:syl]+2,2)...]
%                   ...

spectralShapingFilterLength = size(spectralShapingFilter,2);
if rem(spectralShapingFilterLength,symbolLength) > 0
  fprintf('ERROR: The size of spectralShapingFilter is not a multiple of symbolLength.');
end

% Generate base-band symbol waveforms
matrixSymbolsShapedToSignals = spectralShapingFilter.'*inComplexSequence;

noSymbols    = length(inComplexSequence);
outSignalBB = zeros((noSymbols*symbolLength) + spectralShapingFilterLength,1);

sourceRange = [1:symbolLength];
targetRange = [1:noSymbols*symbolLength];
for symbolIntervalIndex = 1:fix(spectralShapingFilterLength/symbolLength)
    offset = (symbolIntervalIndex-1)*symbolLength;
    targetSignals = matrixSymbolsShapedToSignals(sourceRange + offset,:);
    outSignalBB(targetRange + offset) = outSignalBB(targetRange + offset) + targetSignals(:);
end
```

A.4.6 CreateInverseFilter.m

```
function [inverseIRFs,extraInfo] = CreateInverseFilter(inIRFs,filterType,filterSettings)

filterLength  = size(inIRFs,1);
noChannelsOut = size(inIRFs,3);
noChannelsIn  = size(inIRFs,2);
fprintf('Creating Filter: %-25s',filterType);

switch filterType
  case 'No Filtering'              %=============================================================
    for ii = 1:noChannelsOut
        for jj = 1:noChannelsIn
            inverseIRFs(:,ii,jj) = ii==jj;
        end
    end

  case 'TimeReversal'              %=============================================================
    for ii = 1:noChannelsOut
        for jj = 1:noChannelsIn
            inverseIRFs(:,ii,jj) = conj(inIRFs(filterLength:-1:1,jj,ii))/max(abs(inIRFs(:,jj,ii)));
        end
    end

  case 'Tikhonov IF: Path'         %=============================================================
    % Put the IRF's into the frequency domain...
    for ii = 1:noChannelsOut; s(:,:,ii) = fft([inIRFs(:,:,ii) ]); end

    maxval = max(abs(s(:)));

    for ii_w = 1:filterLength
        if (rem(ii_w, floor(filterLength/50)+1) == 0) fprintf('.'); end
        % C_w- Frequency transfer matrix
        % H_w- Inverse frequency transfer matrix
```

```
33          % Essentially: h(i,j)=(abs(c(j,i))^2+beta*maxval)^(-1)*conj(c(j,i))
34          C_w = squeeze(s(ii_w,:,:));
35          H_w(ii_w,:,:) = [([abs(C_w).'].^2+filterSettings.beta*maxval).^(-1)].*C_w';
36      end
37
38      % Convert the iIRF's into the time domain...
39      for ii = 1:noChannelsIn; this_iIRF(:,:,ii) = ifft(H_w(:,:,ii)); end
40
41      % apply a half temporal shift..
42      inverseIRFs = this_iIRF([floor(filterLength/2)+1:filterLength 1:floor(filterLength/2)],:,:);
43
44      % Normalise the each filters (seperatly)
45      for ii = 1:noChannelsOut
46          for jj = 1:noChannelsIn
47              inverseIRFs(:,ii,jj) = inverseIRFs(:,ii,jj)./max(abs(inverseIRFs(:,ii,jj)));
48          end
49      end
50
51  case 'Tikhonov IF: Channel'     %=============================================
52      % Put the IRF's into the frequency domain...
53      for ii = 1:noChannelsOut; s(:,:,ii) = fft([inIRFs(:,:,ii) ]); end
54
55      maxval = max(abs(s(:)));
56      for ii_w = 1:filterLength
57          if (rem(ii_w, floor(filterLength/50)+1) == 0) fprintf('.'); end
58          % C_w - Frequency transfer matrix
59          % H_w - Inverse frequency transfer matrix
60          C_w = squeeze(s(ii_w,:,:));
61          this_H = [];
62          for jj = 1:noChannelsIn
63              C_row = C_w(jj,:);
64              warning off MATLAB:nearlySingularMatrix
65              this_H = [this_H inv(C_row'*C_row+filterSettings.beta*eye(noChannelsOut)*maxval)*C_row'];
66              warning on MATLAB:nearlySingularMatrix
67          end
68          H_w(ii_w,:,:) = this_H;
69      end
70
71      % Convert the iIRF's into the time domain...
72      for ii = 1:noChannelsIn; this_iIRF(:,:,ii) = ifft(H_w(:,:,ii)); end
73
74      % apply a half temporal shift..
75      inverseIRFs = this_iIRF([floor(filterLength/2)+1:filterLength 1:floor(filterLength/2)],:,:);
76
77      % Then we normalise
78      for jj = 1:noChannelsIn
79          inverseIRFs(:,:,jj) = inverseIRFs(:,:,jj)./max(max(abs(inverseIRFs(:,:,jj))));
80      end
81
82  case 'Tikhonov IF: Full MIMO' %===============================================
83      % Put the IRF's into the frequency domain...
84      for ii = 1:noChannelsOut; s(:,:,ii) = fft([inIRFs(:,:,ii) ]); end
85
86      maxval = max(abs(s(:)));
87
88      for ii_w = 1:filterLength
89          if (rem(ii_w, floor(filterLength/50)+1) == 0) fprintf('.'); end
90          C_w = squeeze(s(ii_w,:,:));   % C(w) - transfer matrix
91
92          warning off MATLAB:nearlySingularMatrix
93          H_w(ii_w,:,:) = inv(C_w'*C_w + filterSettings.beta*eye(noChannelsOut)*maxval)*C_w';
94          warning on MATLAB:nearlySingularMatrix
95      end
96
97      % Convert the iIRF's into the time domain...
```

```
98      for ii = 1:noChannelsIn; this_iIRF(:,:,ii) = ifft(H_w(:,:,ii)); end
99
100     % apply a half temporal shift..
101     inverseIRFs = this_iIRF([floor(filterLength/2)+1:filterLength 1:floor(filterLength/2)],:,:);
102
103     % Then we normalise
104     inverseIRFs = inverseIRFs/max( abs(inverseIRFs(:)) );
105
106   case 'Milica: One-Side Filter' %=============================================================
107     % Put the IRF's into the frequency domain...
108     for ii = 1:noChannelsOut; s(:,:,ii) = fft([inIRFs(:,:,ii) ]); end
109
110     maxval = max(abs(s(:)));
111     gammas = sum(abs(s.^2),3);
112
113     for ii = 1:noChannelsOut
114         for jj = 1:noChannelsIn
115             H_w(:,ii,jj) = (gammas(:,jj) + filterSettings.beta*maxval).^(-1).*conj(s(:,jj,ii));
116         end
117     end
118
119     % Convert the iIRF's into the time domain...
120     for ii = 1:noChannelsIn; this_iIRF(:,:,ii) = ifft(H_w(:,:,ii)); end
121
122     % shift, and normalise.
123     this_iIRF = this_iIRF([floor(filterLength/2)+1:filterLength 1:floor(filterLength/2)],:,:);
124     inverseIRFs =  this_iIRF./max(abs(this_iIRF(:)));
125
126   case 'Milica: Two-Side Filter' %=============================================================
127     % Put the IRF's into the frequency domain...
128     for ii = 1:noChannelsOut; s(:,:,ii) = fft([inIRFs(:,:,ii) ]); end
129
130     maxval = max(abs(s(:)));
131     gammas = sum(abs(s.^2),3);
132
133     for jj = 1:noChannelsIn
134         H_w_0(:,jj) = (gammas(:,jj) + filterSettings.beta*maxval).^(-1/4);
135     end
136
137     for ii = 1:noChannelsOut
138         for jj = 1:noChannelsIn
139             H_w(:,ii,jj) = (gammas(:,jj) + filterSettings.beta*maxval).^(-3/4).*conj(s(:,jj,ii));
140         end
141     end
142
143     % Convert the iIRF's into the time domain...
144     for ii = 1:noChannelsIn; this_iIRF(:,:,ii) = ifft(H_w(:,:,ii)); end
145     this_iIRF_0 = ifft(H_w_0);
146
147     % shift, and normalise.
148     this_iIRF_0 = this_iIRF_0([floor(filterLength/2)+1:filterLength 1:floor(filterLength/2)],:);
149     this_iIRF = this_iIRF([floor(filterLength/2)+1:filterLength 1:floor(filterLength/2)],:,:);
150     inverseIRFs =  this_iIRF./max(abs(this_iIRF(:)));
151     extraInfo.rxFilter = this_iIRF_0./max(abs(this_iIRF_0(:)));
152
153   otherwise
154     fprintf('Sorry!! Don''t know that type of filter...');
155 end
156 fprintf('\n');
```

A.4.7 DetectorAdaptiveMSE.m

```
function [signalError detectedSymbols filterTaps filterTapHistory filteredSignal ...
          feedbackFilterTaps] = DetectorAdaptiveMSE(constellationValues,sampledSignal,delta,...
                                                    filterTaps,trainingSequence,fractionalSpacing,...
                                                    feedbackFilterTaps,filterPeakIndex)
% USAGE: [signalError detectedSymbols filterTaps filterTapHistory filteredSignal ...
%         feedbackFilterTaps] = DetectorAdaptiveMSE(constellationValues,sampledSignal,delta,...
%                                                   filterTaps,trainingSequence,fractionalSpacing,...
%                                                   feedbackFilterTaps,filterPeakIndex)
%
% This function implements the decision-feedback lease mean square (LMS) algorithm that is focused
% on the minimisation of the mean-square-error (MSE). The adaptive LMS algorithm is described in
% greater detail in [Proakis 2001] Section 11.1.2, pages 663-666. The algorithm is implemented as:
%
%   filteredSignal_{k} = \sum_{j=-K_1}^{K_2} c_{j} * sampledSignal_{k-j}
%                      + \sum_{j=0}^{K_d}    d_{j} * detectedSymbols_{k-j}
%
% and then the filter taps are updated according to:
%
%   c_{j} = c_{j} + delta*signalError(k)*conj(sampledSignal_{k-j})
%   d_{j} = d_{j} + delta*signalError(k)*conj(detectedSymbols_{k-j})
%
% where
%
%   signalError_{k} = filteredSignal_{k} - detectedSymbol_{k}
%
% Note that this description varies from Proakis [2001] in that seperate coefficients are used for
% the feedforward, c_{j}, and the feedback, d_{j}, taps and introduces K_2 (which is set to 0 in
% Proakis [2001]). This is because the non "decision-feedback" equaliser has K_d=0 and K_2 != 0.
%
% For this implementation, the vectors, filterTaps and feedbackFilterTaps contains the coefficients:
%
% filterTaps         = [c_{0}  c_{1}    ... c_{K-1} c_{K}  ] for filterPeakIndex = 1
% filterTaps         = [c_{-1} c_{0}    ... c_{K-2} c_{K-1}] for filterPeakIndex = 2
%                      ...
% filterTaps         = [c_{-K} c_{-K+1} ... c_{-1} c_{0}   ] for filterPeakIndex = K
%   and
% feedbackFilterTaps = [d_(1)  d_(2) ...]
%
% where K is filterLength.  before calling this function for the first time, the variables,
% filterTaps should be initialised such that c_{j} = (j==0)?1:0 and feedbackFilterTaps = 0.
%
% Note: To implement a DFE-MSE equaliser, filterPeakIndex with the feedforward only filtering
% the past and current symbols, then set filterPeakIndex = K.

filterLength         = length(filterTaps);
feedbackFilterLength = length(feedbackFilterTaps);
trainingLength       = length(trainingSequence);

sampledSignalZP = [zeros(filterLength-(filterPeakIndex-1)*fractionalSpacing-1,1); ...
                   sampledSignal; ...
                   zeros((filterPeakIndex-1)*fractionalSpacing,1)];

detectedSymbols = zeros(1, feedbackFilterLength);

sampleIndexFSHistory = [];
sampleIndex = 0;

% Cycle through all fractional stepped values in input signal
for sampleIndexFS = 1:length(sampledSignalZP)-filterLength + 1;
    if mod(sampleIndexFS-1,fractionalSpacing) == 0
        sampleIndexFSHistory = [sampleIndexFSHistory sampleIndexFS];
        sampleIndex = sampleIndex + 1;

```

```
64          % Apply filter
65          if (feedbackFilterLength > 0)
66              filteredSignal(sampleIndex) = ...
67                  filterTaps*sampledSignalZP([filterLength:-1:1]+sampleIndexFS-1) ...
68                  + feedbackFilterTaps*detectedSymbols(sampleIndex-1+[feedbackFilterLength:-1:1]).';
69          else
70              filteredSignal(sampleIndex) = ...
71                  filterTaps * sampledSignalZP([filterLength:-1:1]+sampleIndexFS-1);
72          end
73
74          filterTapHistory(:,sampleIndex) = [filterTaps feedbackFilterTaps];
75
76          % Detect current symbol or use training symbol
77          if (sampleIndex > trainingLength)
78              [dummy id] = min(abs(filteredSignal(sampleIndex)-constellationValues));
79              detectedSymbols(sampleIndex+feedbackFilterLength) = constellationValues(id);
80          else
81              detectedSymbols(sampleIndex+feedbackFilterLength) = trainingSequence(sampleIndex);
82          end
83          % Calculate the error
84          signalError(sampleIndex) = ...
85              detectedSymbols(sampleIndex+feedbackFilterLength) - filteredSignal(sampleIndex);
86
87          % Update the filter taps
88          filterTaps = filterTaps ...
89              + delta*signalError(sampleIndex)*sampledSignalZP([filterLength:-1:1].'+sampleIndexFS-1)';
90          if (feedbackFilterLength)
91              feedbackFilterTaps = feedbackFilterTaps ...
92                  + delta * signalError(sampleIndex) ...
93                  * conj(detectedSymbols([feedbackFilterLength:-1:1].'+sampleIndex-1));
94          end
95      end
96  end
97  detectedSymbols = detectedSymbols(feedbackFilterLength+1:end);
```

A.4.8 DetectorAdaptiveRLS.m

```
1   function [signalError detectedSymbols filterTaps filterTapHistory filteredSignal ...
2             feedbackFilterTaps] = DetectorAdaptiveRLS(constellationValues,sampledSignal,delta, ...
3                                                       filterTaps,trainingSequence,fractionalSpacing, ...
4                                                       feedbackFilterTaps,filterPeakIndex)
5   % USAGE: [signalError detectedSymbols filterTaps filterTapHistory filteredSignal ...
6   %         feedbackFilterTaps] = DetectorAdaptiveRLS(constellationValues,sampledSignal,delta, ...
7   %                                                   filterTaps,trainingSequence,fractionalSpacing, ...
8   %                                                   feedbackFilterTaps,filterPeakIndex)
9   %
10  % This function implements the recursive least square (RLS) algorithm that is described in [Proakis
11  % 2001] Section 11.4.1, pages 683-686. The equations describing the algorithm are to extensive to
12  % include here.
13  %
14  % For this implementation, the vectors, filterTaps and feedbackFilterTaps contains the coefficients:
15  %
16  % filterTaps          = [c_{0}  c_{1}     ... c_{K-1} c_{K}   ] for filterPeakIndex = 1
17  % filterTaps          = [c_{-1} c_{0}     ... c_{K-2} c_{K-1}] for filterPeakIndex = 2
18  %                         ...
19  % filterTaps          = [c_{-K} c_{-K+1} ... c_{-1} c_{0}    ] for filterPeakIndex = K
20  %   and
21  % feedbackFilterTaps = [d_(1)  d_(2) ...]
22  %
23  % where K is filterLength.  before calling this function for the first time, the variables,
24  % filterTaps should be initialised such that c_{j} = (j==0)?1:0 and feedbackFilterTaps = 0.
25  %
26  % Note: To implement a DFE-MSE equaliser, filterPeakIndex with the feedforward only filtering
```

```
% the past and current symbols, then set filterPeakIndex = K.

filterLength          = length(filterTaps);
feedbackFilterLength = length(feedbackFilterTaps);
trainingLength        = length(trainingSequence);

sampledSignalZP = [zeros(filterLength-(filterPeakIndex-1)*fractionalSpacing-1,1); ...
                   sampledSignal; ...
                   zeros((filterPeakIndex-1)*fractionalSpacing,1)];

detectedSymbols = zeros(1,feedbackFilterLength);

C = [filterTaps feedbackFilterTaps].';
P = eye(length(C));

sampleIndexFSHistory = [];
sampleIndex = 0;
for sampleIndexFS = 1:length(sampledSignalZP) - filterLength + 1;
    if mod(sampleIndexFS-1,fractionalSpacing) == 0
        sampleIndexFSHistory = [sampleIndexFSHistory sampleIndexFS];
        sampleIndex = sampleIndex + 1;

        % Define the vector of samples used to perform the filtering on
        if (feedbackFilterLength)
            Y = [sampledSignalZP([filterLength:-1:1] + sampleIndexFS - 1); ...
                 detectedSymbols(sampleIndex - 1 + [feedbackFilterLength:-1:1]).'];
        else
            Y = [sampledSignalZP([filterLength:-1:1] + sampleIndexFS - 1)];
        end

        % Perform the filtering for the current sample
        filteredSignal(sampleIndex) = Y.'*C;
        filterTapHistory(:,sampleIndex) = C.';

        % Detect current symbol or use training symbol
        if (sampleIndex > trainingLength)
            [dummy id] = min(abs(filteredSignal(sampleIndex)-constellationValues));
            detectedSymbols(sampleIndex+feedbackFilterLength) = constellationValues(id);
        else
            detectedSymbols(sampleIndex+feedbackFilterLength) = trainingSequence(sampleIndex);
        end

        % Calculate the error
        signalError(sampleIndex) = ...
            detectedSymbols(sampleIndex+feedbackFilterLength) - filteredSignal(sampleIndex);

        if (delta > 0)
            % Compute the Kalman gain vector
            K = P*conj(Y)/(delta + Y.'*P*conj(Y));

            % Update the inverse of the correlation matrix
            P = (1/delta)*(P - K*Y.'*P);

            % Update the filter taps
            C = C + K*signalError(sampleIndex);
        end
    end
end

% Retrieve the filter taps for returning to the calling function.
filterTaps = C(1:filterLength).';
feedbackFilterTaps = C(filterLength + [1:length(feedbackFilterTaps)]).';

detectedSymbols = detectedSymbols(feedbackFilterLength+1:length(detectedSymbols));
```

A.4.9 DetectorAdaptiveZF.m

```
function [signalError detectedSymbols filterTaps filterTapHistory filteredSignal] = ...
    DetectorAdaptiveZF(constellationValues,sampledSignal,delta,filterTaps,trainingSequence)
% USAGE: [signalError detectedSymbols filterTaps filterTapHistory filteredSignal] = ...
%             DetectorAdaptiveZF(constellationValues,sampledSignal,delta,filterTaps,trainingSequence)
%
% This function implements the zero forcing (ZF) algorithm that is described in greater detail in
% [Proakis 2001] Section 11.1.1, pages 661-662.  The algorithm is implemented as:
%
%   filteredSignal_{k} = \sum^{J}_{j=0} filterTaps_{j}*sampledSignal_{k-j}
%
% and the filter taps are updated according to:
%
%   filterTaps^{k+1}_{j} = filterTaps^{k}_{j} + delta * signalError_{k} * conj(detectedSymbols_{k-j})
%
% where
%
%   signalError_{k} = filteredSignal_{k} - detectedSymbol_{k}
%

filterLength   = length(filterTaps);
trainingLength = length(trainingSequence);

% Zero Pad sampledSignal
sampledSignalZP = [zeros(filterLength,1); sampledSignal];
detectedSymbols = zeros(1,filterLength);

% Cycle through all symbols in input signal
for sampleIndex = 1:length(sampledSignalZP)-filterLength;

    % Perform the filtering for the current sample
    filteredSignal(sampleIndex) = filterTaps*sampledSignalZP(sampleIndex+[filterLength:-1:1]);
    filterTapHistory(:,sampleIndex) = filterTaps;

    % Detect current symbol or use training symbol
    if (sampleIndex > trainingLength)
        [dummy id] = min(abs(filteredSignal(sampleIndex)-constellationValues));
        detectedSymbols(sampleIndex+filterLength) = constellationValues(id);
    else
        detectedSymbols(sampleIndex+filterLength) = trainingSequence(sampleIndex);
    end

    % Calculate the error
    signalError(sampleIndex) = detectedSymbols(sampleIndex+filterLength)-filteredSignal(sampleIndex);

    % Update the filter taps
    filterTaps = filterTaps + ...
        delta*signalError(sampleIndex)*conj([detectedSymbols(sampleIndex+[filterLength:-1:1])]);
end

detectedSymbols = detectedSymbols(filterLength+1:length(detectedSymbols));
```

A.4.10 DetectorNonAdaptive.m

```
function [signalError detectedSymbols constellationValues] = ...
    DetectorNonAdaptive(modulationMethod,sampledSignalComponents,BasisInfo,modulationSettings)
% USAGE: [signalError detectedSymbols constellationValues] = ...
%     DetectorNonAdaptive(modulationMethod,sampledSignalComponents,BasisInfo,modulationSettings)
%
% This performs symbol detection on the sampled signal without the use of an adaptive equaliser.

constellationValues = GetConstellationValues(modulationMethod,BasisInfo,modulationSettings);

constellationValueComponents(:,1) = real(constellationValues);
constellationValueComponents(:,2) = imag(constellationValues);

% Perform the euclidean distance calculations:
for (sampleIndex = 1:size(sampledSignalComponents,1))
    for constelValueIndex = 1:size(constellationValueComponents,1)
        constellationErrors(sampleIndex,constelValueIndex) = ...
            sqrt(sum([...
                constellationValueComponents(constelValueIndex,:)-sampledSignalComponents(sampleIndex,:) ...
                ;].^2));
    end
end
[signalError detectedSymbols] = min(constellationErrors,[],2);
```

A.4.11 FindSignalFirstPeak.m

```
function [peakIndex peakValueInverse] = FindSignalFirstPeak(inputSignal,symbolLength)
% USAGE: [peakIndex peakValueInverse] = FindSignalFirstPeak(inputSignal,symbolLength)
%
% This function returns the largest peak within 2 symbolLength's after the signal goes above 0.3
% times the maximum level of the signal.

inputSignal = inputSignal(:);

peakSearchRangeStartIndex = min(find(abs(inputSignal) > 0.3*max(abs(inputSignal))));
peakSearchRangeEndIndex   = min(peakSearchRangeStartIndex + 2*symbolLength,length(inputSignal));

[dummy peakIndex] = max(abs(inputSignal(peakSearchRangeStartIndex:peakSearchRangeEndIndex)));
peakIndex = peakSearchRangeStartIndex + peakIndex - 1;

peakValueInverse = inputSignal(peakIndex).^-1;
```

A.4.12 GetConstellationValues.m

```
function constellationValues = GetConstellationValues(modulationMethod,BasisInfo,modulationSettings)
% USAGE: constellationValues = GetConstellationValues(modulationMethod,BasisInfo,modulationSettings)
%
%  This function creates a complex vector that containing the complex values of the constellation
% for a modulation method.
%

if strcmp(modulationMethod,'DSB_PAM')
    k = modulationSettings.k;
    M = 2^k;

    greyEncodeMap = GreyEncodeMap(k);
    complexSymbolVectors = (greyEncodeMap*2 - M + 1)./(M-1)*sqrt(BasisInfo.E_g/2);
    constellationValues  = complexSymbolVectors.';

```

```
elseif strcmp(modulationMethod,'PSK')
    k = modulationSettings.k;
    M = 2^k;

    greyEncodeMap = GreyEncodeMap(k);
    complexSymbolVectors = exp(j*2*pi*(greyEncodeMap-1)/M + j*modulationSettings.addphase);
    constellationValues  = complexSymbolVectors.';

elseif strcmp(modulationMethod,'QAM')
    complexSymbolVectors = bbgenstruc.constellation(1:M)';
    constellationValues  = complexSymbolVectors.';

else
    error(sprintf('Modulation method, %s, not implemented\n',modulationMethod));
end
```

A.4.13 GreyDecodeMap.m

```
function greyDecodeMap = GreyDecodeMap(M)
% USAGE: greyDecodeMap = GreyDecodeMap(M)
%
% This function returns the map to obtain the original sequence of numbers from a grey-encoded
% numerical sequence.
%
% See the help for GreyEncodeMap for more information, and an example usage.

[a,greyDecodeMap] = sort(GreyEncodeMap(M));
greyDecodeMap     = greyDecodeMap - 1;
```

A.4.14 GreyEncodeMap.m

```
function greyEncodeMap = GreyEncodeMap(M)
% USAGE: greyEncodeMap = GreyEncodeMap(M)
%
% This function generates a mapping so that numbers are mapped to binary numbers such that the
% variations between adjacent binary number contain only a single bit fluctuation:
%
% i.e.
%   0 -> 000    4 -> 111
%   1 -> 001    5 -> 110
%   2 -> 010    6 -> 101
%   3 -> 011    7 -> 100
%
% A simple algorithm for M bits was derived from the example presented in [Proakis 2001], Page 171.
% The algorithm involves using the previous map (M-1), and appending the previous map in reverse
% order, and the M'th bit to 1.
%
% The condition that only a single bit change occurs between each bit can easily be verified using
% Mathematical Induction. (Given the previous map satisfies the condition, that the reversal of the
% order of the map with a fixed additional bit also satisfies the condition; andthe bits for the
% last value in the first half and the first value in the second half only contain a single bit
% change, then given that the map for M=1 satisfies the condition, then the map for M>1 must also
% satisfy the condition.)
%
% EXAMPLE USAGE:
%
%       greyEncodeMap = GreyEncodeMap(M);
%       greyMappedNumericalSequence = greyEncodeMap(originalNumericalSequence + 1);
%
%       %... transmission / reception...
```

```
%
%       greyDecodeEncodeMap = GreyEncodeMap(M);
%       receivedNumericalSequence = greyDecodeEncodeMap(greyMappedNumericalSequence + 1);
%
%   where originalNumericalSequence contains values between 0 and M,
%         greyMappedNumericalSequence is the mapped sequence of values (also between 0 and M),
%
%   Note that the '+ 1' is required since MATLAB indexes from 1 rather than 0.

greyEncodeMap = [0 1];
for ii = 1:(M-1);
    greyEncodeMap = [greyEncodeMap (greyEncodeMap(end:-1:1)+2^ii)];
end
```

A.4.15 MatrixConvolve.m

```
function matrixC = MatrixConvolve(matrixA,matrixB)
% USAGE: matrixC = MatrixConvolve(matrixA,matrixB)
%
%  This function performs the matrix impulse response convolution.
%
%   [ matrixC ] = [ matrixA ] [ matrixB ]
%
% where matrixC, matrixA and matrixB are 3-dimensional matricies. The first dimension being time and
% the second and third dimensions those that get multiplied in normal matrix. The calculations
% performed by this function can be expressed in latex math notation as:
%
%  c(:,i,j) = \sum_t ( a(:,i,t) * b(:,t,j) )

if (size(matrixA,3) ~= size(matrixB,2))
    error('Sorry, the dimensions don''t add up');
end

matrixC = zeros(size(matrixB,1)+size(matrixA,1)-1,size(matrixA,2),size(matrixB,3));

for i_m = 1:size(matrixA,2)
    for i_n = 1:size(matrixB,3)
        for i_t = 1:size(matrixA,3)
            matrixC(:,i_m,i_n) = matrixC(:,i_m,i_n) + conv(matrixA(:,i_m,i_t),matrixB(:,i_t,i_n));
        end
    end
end
```

A.4.16 PassbandToBaseband.m

```
function outIRFsBB = PassbandToBaseband(inIRFs,freqSampling,freqCarrier)
% USAGE: outIRFsBB = PassbandToBaseband(inIRFs,freqSampling,freqCarrier)
%
% This function uses the fft function to shift impulse responses from passband to baseband.
% [Proakis 2001] page 153, Eq. 4.1-26 defines that:
%
%   H_l(f-f_c) = { H(f)          for f > 0
%                { 0             for f < 0
%
%  where H is the frequency response of the impulse responses. Setting
%
%    fdash = f-f_c
%
%  this becomes:
%
```

```
16 %   H_l(fdash) = { H(fdash+f_c)  for fdash > f_c
17 %                 { 0             for fdash < f_c
18 %
19 % Now the fft and ifft function convert a time-based signal into a frequency based signal according
20 % to:
21 %
22 %    0              1            ... noSamples/2
23 %   [H(0) H(f_s/N) H(2*f_s/N)  ... H((N/2-1)*f_s/N)  ...
24 %
25 %    1+noSamples/2 2+noSamples/2       ... noSamples
26 %    H(-f_s/2)     H(-(N/2-1)*f_s/N)  ... H(fs/N)]
27 %
28 % where
29 %       N is the number of time-samples being converted
30 %       f_s is the sample rate.
31 %
32 % Using the above relationship between H_l(f) and H(f), the values are mapped as follows:
33 %
34 %                                          f_s/2-f_s/N
35 %                          f_c              |
36 %                           |               |
37 % H(f)    [++++++++++++++++++++++++++++++++++-------------------------]
38 %           |---- R2 ----||------- R1 -------|
39 %           |            |i1                i2
40 %           i3           i4
41 %
42 %
43 % H_l(f) [++++++++++++++++++++++++++++++++++-------------------------]
44 %           |------- R1 -------|  0 . . . . . . . . . . . . . . 0 |---- R2 ----|
45 %           i1                 i2                                 i3           i4
46 %
47
48 noSamples = size(inIRFs,1);
49
50 index1 = ceil(freqCarrier/freqSampling*noSamples);
51 index2 = noSamples/2;
52
53 index3 = 1;
54 index4 = index1 - 1;
55
56 noZeros = noSamples - (index2 - index1 + 1) - (index4 - index3 + 1);
57
58 for ii = 1:size(inIRFs,3)
59     for jj = 1:size(inIRFs,2)
60         fftIRFs   = fft(inIRFs(:,jj,ii));
61         fftIRFsBB = [fftIRFs(index1:index2) ; zeros(noZeros,1) ; fftIRFs(index3:index4)];
62         outIRFsBB(:,jj,ii) = ifft(fftIRFsBB);
63     end
64 end
```

A.4.17 RaisedCosineFrequencySpectrum.m

```
function outSpectrum = RaisedCosineFrequencySpectrum(frequencyValues,symbolPeriod,beta)
% USAGE: outSpectrum = RaisedCosineFrequencySpectrum(frequencyValues,symbolPeriod,beta)
%
%  This function returns the frequency spectrum for a raised-cosine waveform. The formulas used are
%  those given by [Proakis 2001] on page 560, Equation. 9.2-29:
%
%       X_{rc}(f) = { T                                    for          0 <= |f| <= (1-beta)/(2T)
%                   { (T/2)(1+cos((phi*T/beta)*(|f| - ((1-beta)/(2T)))))
%                                                       for (1-beta)/(2T) <  |f| <= (1+beta)/(2T)
%                   { 0                                   for (1+beta)/(2T) <  |f|
%
%
for freqIdx = 1:length(frequencyValues)
    frequency = frequencyValues(freqIdx);
    if  abs(frequency)>= 0 && abs(frequency) < (1-beta)/(2*symbolPeriod)
        outSpectrum(freqIdx) = symbolPeriod;

    elseif (1-beta)/(2*symbolPeriod)<=abs(frequency) && abs(frequency)<=(1+beta)/(2*symbolPeriod)
        outSpectrum(freqIdx) = ...
            (symbolPeriod/2) ...
            * ( 1 + cos((pi*symbolPeriod/beta)*(abs(frequency)-(1-beta)/(2*symbolPeriod)) ) ...
              );

    else
        outSpectrum(freqIdx) = 0;

    end
end
```

A.4.18 ResampleIRFs.m

```
function outIRFs = ResampleIRFs(inIRFs,freqTo,freqFrom)
for ii = 1:size(inIRFs,2)
    for jj = 1:size(inIRFs,3)
        outIRFs(:,ii,jj) = ...
            resample(real(inIRFs(:,ii,jj)),freqTo,freqFrom) ...
            + i * resample(imag(inIRFs(:,ii,jj)),freqTo,freqFrom);
    end
end
```

A.4.19 SignalPhaseEstimatorPassbandToBaseband.m

```
function [basebandSignal maxValIdx phiPeak gain] = ...
    SignalPhaseEstimatorPassbandToBaseband(receivedSignal,freqCarrier,freqSampling,lowPassFilter,...
                                           symbolLength)
% USAGE: [basebandSignal maxValIdx phiPeak gain] = ...
%   SignalPhaseEstimatorPassbandToBaseband(receivedSignal,freqCarrier,freqSampling,lowPassFilter,...
%                                          symbolLength)
%
% This function takes a passband signal that is the response of a signal resulting from a baseband
% signal commencing with a value of '1'. The received passband signal is converted to baseband, and
% the first peak is used to perform a phase estimate, phiPeak, for the entire received signal. This
% function also returns the index of the maximum peak in order to provide a form of synchronisation.
%

receivedSignalLength = size(receivedSignal,1);

receivedSignalSin = (receivedSignal'.*sin(2*pi*freqCarrier/freqSampling*[0:(receivedSignalLength-1)]));
```

```
17 receivedSignalCos = (receivedSignal'.*cos(2*pi*freqCarrier/freqSampling*[0:(receivedSignalLength-1)]));
18
19 basebandSignal = conv(receivedSignalCos+i*receivedSignalSin,lowPassFilter);
20
21 % To find the peak, set an index at a trigger when signal first goes above 0.4*max peak in signal
22 % then find peak level in 0:symbolLength following this point.
23
24 absSignal            = abs(basebandSignal);
25 triggerIdx           = min(find(absSignal > 0.4 * max(absSignal)));
26 [maxVal maxValIdx] = max(absSignal(triggerIdx + [0:symbolLength]));
27 maxValIdx            = triggerIdx + maxValIdx - 1;
28
29 gain           = abs(1/basebandSignal(maxValIdx));
30 phiPeak        = angle(basebandSignal(maxValIdx));
31 basebandSignal = basebandSignal/basebandSignal(maxValIdx);
```

B Figure Attributions

This thesis includes figures that are artwork created by other people or a modified / digitised version thereof. For the following figures, attribution is as follows:

Figure 3.9 *"Modern french double horn in F/B-flat and Kruspe valve ordering (Besson BE 702), seen from back"*, Hk kng `http://commons.wikimedia.org/wiki/File:French_Horn_back.svg`. Licensed under the Creative Commons Attribution ShareAlike 3.0 Licence.

Figure 3.9 *"An icon from the GNOME-icon-theme",* 2007 GNOME icon artist, `http://commons.wikimedia.org/wiki/File:Gnome-multimedia-player.svg`. Licensed under the GNU General Public License.

Figure 5.14 Images of computer and server adapted from the Tango icon library `http:///tango.freedesktop.org/Tango_Icon_Library`. The Tango base icon theme is released to the Public Domain.