University of Southern Queensland

Faculty of Engineering & Surveying

Simulation and Analysis of MIMO-OFDM for a 4G Cellular Network

A dissertation submitted by

Sarah Hugo

in fulfilment of the requirements of

ENG4112 Research Project

towards the degree of

Bachelor of Engineering (Computer Systems) & Bachelor of Information Technology (Applied Computer Science)

Submitted: October, 2009

Abstract

In the disciplines that study communications, there have been increasing interest in ways and means to improve the technology behind the mobile phone. This interest extends from the antennas used to broadcast the signals to the modulation techniques used on the signals themselves.

The theory is that the antennas can be improved by applying a technique known as MIMO, or multiple-input multiple-output, and the signals can be improved by the modulation technique of OFDM, or orthogonal frequency divisional multiplexing. This concept, as a whole, is known as MIMO-OFDM, and will hopefully be used as the basis of the 4G cellular network in coming years. In developing these, however, it also means covering a wide of topics such as signal interference, channel capacity, fading, et cetera.

The aim of this thesis is to analyze this MIMO-OFDM concept, and all it entails, to determine if it is indeed possible to be used as the basis of a cellular network. This will be by means of a computer simulation in C/C++.

University of Southern Queensland

Faculty of Engineering and Surveying

ENG4111 Research Project Part 1 & ENG4112 Research Project Part 2

Limitations of Use

The Council of the University of Southern Queensland, its Faculty of Engineering and Surveying, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its Faculty of Engineering and Surveying or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course "Project and Dissertation" is to contribute to the overall education within the student's chosen degree programme. This document, the associated hardware, software, drawings, and other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.

Frank Bullen

Professor Frank Bullen
Dean
Faculty of Engineering and Surveying

Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.

I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

Sarah Hugo

0031141187

SA Hugo

Signature

29 October 2009

Date

Acknowledgments

This thesis was typeset using LaTeX 2ε .

I would like thank my family for their support during the long process of researching, programming, and writing this document, both verbal and non-verbal. I would also like to thank my supervisor, for his guidance, support, and wisdom. My thanks also to all the researchers whose words I quote and reference.

On their shoulders I aspire to stand.

Sarah Hugo

University of Southern Queensland
October 2009

Contents

List of Figures

List of Tables

Nomenclature

f_s The minimum sampling frequency (see Nyquist)

ACMA Australian Communications and Media Authority.

aliasing Errors in sampling that sampling that make the signal ‘appear’ faster than it truly is.

Antenna A device radiating signals, with its transmission and receiving happening effectively side by side.

antenna array A number of antennas grouped together, in a certain pattern

attenuation A reduction in the amplitude of the signal, due to fading.

base station The antenna at the centre of a cell.

beamforming A directional antenna array’s signal is concentrated in one area.

BER Bit Error Rate

bit error rate The rate at which errors are received during a transmission.

BS Base Station

cell handover The process where a mobile moves from one cell to another

cell radius The distance signals from the BS can be received by a mobile phone

channel The transmission medium — air, for a cellular system — the signal exists within.

combination unit antenna An antenna where the transmission and receiving effectively take place on the one antenna.

container A C++ holder object that stores a collection of other objects, or data, and provides with itself a number of additional functions.

delay The time interval between the signal’s propagation and its reception, caused by fading.

deque Double-headed queue, a C++ container.

fading Signal loss due to attenuation, delay, and phase shift

Gaussian fading Interference that happens when there is a clear LOS to the transmitter.

GoS Guarantee of Service

LOS line of sight

MACON Multple Antenna Channl [with] Ofdm and Noise’ system

MACS Multiple Antennas [with] Channels [and] Signals

MIMO Multiple Input Multiple Output

mobile station Another term for the mobile phone

MOCA ’MIMO-OFDM, Channel, Antenna’ system

MS Mobile Station

multipath propagation Fading, and interference, due to signal effects.

Nyquist interval The relationship between the sampling interval and

Nyquist rate The minimum sampling frequency at which a signal can be sampled without introducing errors.

OFDM Orthogonal Frequency Division Multiplexing

permittivity How well a medium transmits an electric field.

phase shift A change in the signal’s relationship with time, due to fading

Rayleigh fading Interference with little or no LOS to the transmitter and heavy cancelation of the signal.

Ricean fading Interference with partial LOS and partial cancelation of the signal.

sampling The process of measuring a signal at regular intervals in order to convert into digital form.

sampling theorem The minimum sampling frequency must be at least twice that of the highest frequency component present in the original signal.

signal An electromagnetic wave

signal model A mathematical model and the resulting equations.

SISO Single Input Single Output

STL Standard Template Library

Chapter 1

Introduction

Communication. It is 'the art or act of transmitting concepts, knowledge, and/or information from one person to another by some means'. (Webster 2009)

It is something fundamental to human nature, and the lifeblood of our species. Ever since the first scratching in the sand and the first cave paintings, mankind has been seeking more efficient means to pass on concepts. In ever increasing ways, communication and the way it is carried out have become part of our way of life. In this modern age, one of the more common means of passing information from one person to another is via the mobile phone, also known as the cell phone.

Unfortunately, anecdotal evidence proves that there is still much frustration with current mobile services, despite upgrades in services and technology. Subscribers experience 'drop-outs' during calls, lack of service and signal in certain areas, and so forth.

For instance, in countries like Australia, it is often necessary to remind inexperienced travelers into the 'Outback' that cellular coverage often dies a few miles outside of most major towns. (Agar 2003)

In the disciplines that study communications, there has been therefore increasing interest in the ways and means to improve the technology behind the mobile phone. Link reliability, spectral efficiency, interference, and channel capacity limits are just a few of the problems that mobile communications systems face. All these areas will be discussed further in this paper. For now, though, here is a brief overview of these topics.

Link reliability is basically the stability of the 'link' between the transmitter and the receiver — which, for the purposes of this paper, are the mobile phone and the base-station respectively.[1] This leads to the terms *uplink*, relating mobile phone to base-station signal (transmission), and *downlink*, relating base-station to mobile phone signal (receiving). (Goldsmith, Jafar, Jindal & Vishwanath 2003)

Spectral efficiency is a relatively broad term, in some respects, but of great importance. It is a measure of the efficiency of the antenna scheme, in particular the signals being sent between the antennas. Without going into too many details, spectral efficiency measures "channel capacity" of the antenna system — be it a single antenna system, or as is increasingly common, a multiple antenna system.

Channel capacity describes the the amount of "information" that can be put into a signal (which has been called a *channel*). Moreover, there are multiple theoretic definitions of the limit of the information these channels (known as *channel capacity*). (Goldsmith et al. 2003). These terms will all be discussed in more detail in Chapter 2.

Interference (also described more fully in Chapter 2) is the technical term for what is commonly known as "noise".

However, it is one of the relatively new concepts in the world of signal processing and communications that has, more and more, aroused intense interest and investigation: the combined theory known as MIMO-OFDM. It is this concept that is the focus of this thesis.

The difference between the current system (single-antenna, or "single-user") and multiple-antenna (or "multiple-user") will be discussed more fully in Chapter 2 and Chapter 4 (see Section 1.2 for details).

[1] Note the emphasis on the user side — signals are transmitted and received according to the user, not the base-station.

1.1 Aims and Objectives

The intense interest in the MIMO-OFDM concept stems from the way it combines the antennas used to broadcast the signals to the modulation techniques used on the signals themselves. The theory is that the antennas can be improved by applying a technique known as MIMO, or *multiple-input multiple-output*, and the signals can be improved by the modulation technique of OFDM, or *orthogonal frequency divisional multiplexing*. Basically, the MIMO technique involves using multiple antennas at the receiver and transmitter, the OFDM modulation technique involves modulating multiple subcarriers into one signal. It is this concept as a whole that is known as MIMO-OFDM.

The reason why these techniques are so interesting is the promise of increased data rates from the OFDM component and increased antenna gain from the MIMO component. If all goes to plan, it will remove — or at least alleviate — many of the problems facing current cellular technology. As such, it is hoped in many circles that MIMO-OFDM will be able to be the core technology for the 4G cellular network in coming years.

It is therefore the aim of this thesis to analyze this MIMO-OFDM concept and determine if it is indeed possible to be used as the basis of a cellular network.

1.2 Overview of the Dissertation

The thesis is organised with the following chapters.

Chapter 2 attempts to provide a brief cover of the background material and the need for the thesis, which has included a wide range of topics. Antennas, multipath propagation (also known as multipath interference), the various forms this interference can take, fading, channel capacity, are all considered (at least a little) in this chapter.

Chapter 3 considers the OFDM system used for the thesis in detail. OFDM is, at its heart, a system of signal modulation. Leading directly on from the previous chapter's topics, this chapter considers the OFDM system in depth, particularly as it relates to channels and signal transmission.

Chapter 4 considers the MIMO antenna-array system in depth. It gives some detail on basic characteristics of antennas, and how this contributes to understanding how the MIMO theory works.

Chapter 5 presents the full concept of MIMO-OFDM, and the inter-relation of the previously mentioned concepts. This chapter lays the foundation for the simulation, as it relates terms from various areas and demonstrates simply how they can be used to develop a cohesive whole — the simulation itself.

Chapter 6 presents the implementation of the simulation in MATLAB, how it was actually programmed, and the advantages and the pitfalls thereof. There were many challenges to overcome, in developing a simulation that covered so many topics, not the least of which was the drawbacks of the languages chosen. All such challenges had to be overcome, and this chapter shows the development process. It also presents the final implementation, and considers the highlights of how it was accomplished, before making recommendations for further work in program development.

Chapter 7 discusses a critical analysis of the simulation's results, and analyzes the success of the program of the program as a simulation of a cellular network. It also presents a performance analysis of the simulation and how it performaned, using standard trace programs. Taking into this into account, it then analyzes how MIMO-OFDM can be applied in a cellular network.

Chapter 8 provides the conclusions and recommendations. Starting with the results of the simulation, both advantages and limits of the MIMO-OFDM concept are emphasized. This chapter also presents the areas for further research and development.

Chapter 2

Background

As was noted briefly in Chapter 1, there are many problems facing the area of mobile communications — which in itself covers many disciplines, but one of the more broader ones is signal processing (SP) and information theory.

There has been much research to define exactly what marks a channel and defines its limits. The landmark work in this area was the thesis proposed by Claude Shannon (1948), which has been said to be the father of the area of information theory itself.[1]

Among other things, Shannon's work described many things that had effect on the channel (transmission medium) which were later built upon, from fading, to interference, and channel capacity.

Before discussing these, though, it should be established exactly what constitutes a mobile phone, in engineering terms.

[1] Incidentally, he wrote an earlier masters' thesis where he proposed the connection between boolean logic and electrical circuitry. That thesis is said to have started digital circuit design. He also developed work on cryptography.

2.1 Mobile Communications

There have been many books written on mobile phones, their history, usage, and their basic concepts. It is not the purpose of this dissertation to go into the details of all these ideas, for this would require the rewriting of said books — and of this there would be no end.

Instead, a brief overview of the main ideas of some of said books and the conducted research will suffice.

2.1.1 Terminology

There is often a gap between the lay person's concepts and terms, and the engineer's terms and concepts. It is no different for mobile phones.[2]

The common idea of a carrier is of that the mobile service provider (MSP), or network providers (NP). This is the name that is displayed on the screen when the mobile is activated (on). Whether the phone has "signal" (coverage) is as simple as looking at "bars" on the screen displaying signal strength. Where the phone "is" is also displayed on the screen beneath the carrier's name, although this is generally depended on signal strength — if there is no signal available, then there is no location displayed.

For most people, that is all that they need to know about how their phone operates.

In truth, network providers (NP) obtain their coverage by purchasing antennas off the manufactures to provide coverage. (See Section 2.2.3 for further details on the cellular concept.) Moreover, in engineering terms, the *carrier* is actually a modulated radio wave to carry a signal. (See also Section 2.2.1 for details on radio waves as relating to mobile phones.)

What is this "signal"?

It should be noted that, depending on the discipline, the term "signal" can have many

[2] Terms are covered here in brief. See later sections, as referenced, for further details.

connotations, or meanings. However, for the purposes of this dissertation, a signal is a means of carrying information from point A to point B, and the means used is an electromagnetic (EM) wave. The amount of energy in this EM wave is one way that this ‘information’ is measured. However, for the purposes of this document, what types of information are present is of limited value. Simply the fact that information of some sort is there is enough.

In addition, just as with a signal, the term ’channel’ can have different connotations, depending on the discipline — and not all of these definitions will agree. In writing this dissertation, however, there can only be one definition for consistency, and sanity. In this document, the channel is simply the transmission medium — in a mobile (cellular) system, that medium is air. (See Section 2.4.1 for further details.) In this paper, therefore, a channel is not a signal, nor is it used in this paper to refer to refer to a band of frequencies. Channels, signals, and frequencies (and bands of said frequencies) will be held as distinctly separate terms.

Signal strength, therefore, refers to the strength of the actual signal received by the mobile, or that transmitted by to be received by said mobile.

Mennen (2005) notes that the phenomena of dropped calls occurs due to both poor coverage (network related issued) and phone-related issues. Some phone-related issues, such as batteries, being unable to reach the NP, et cetera are beyond the scope of the dissertation. For such problems, one should really seek advice from their local retailer or network provider.

Network-related problems, however, often come down to the basic issues that have plagued the cellular networks since their inception, and indeed, were the very reason for their creation: frequency re-use.

2.2 Frequency, EM, and the Cellular Concept

Use of frequency spectrum — that is, the electromagnetic spectrum — has long been a much desired item. The spectrum is cluttered in most western worlds. Modern con-

Table 2.1: Part of the electromagnetic spectrum

Region	**Wavelength** (Angstroms)	**Wavelength** (centimeters)	**Frequency** (Hz)	**Energy** (eV)
Radio	$> 10^9$	> 10	$< 3 \times 10^9$	$< 10^{-5}$

sumers and customers range from from military, to commercial, scientific to modern appliances...not the least of which is the modern mobile phone, which itself came somewhat late into the picture. Finding space for the cellular phone was not an easy task, but it was made easier by the 'frequency reuse' nature of the device. (Agar 2003)

2.2.1 The Mobile 'Radio'

Mobile phones in Australia initially started at 900 MHz. However, due to spectral reuse issues, they also started using the 1800 MHz — phones that use both bands are known as 'dual-band' phones. At the time of writing, there are phones available that use three frequency bands, these are 'tri-band' phones and are generally the more modern phones. The '3G' phones are technically known as double-mode phones, as they take advantage of both the GSM system (one mode) and the WCDMA system (the other mode) for greater coverage.

However, this does not mean that Australia (and other GSM countries) have developed phones that will work in every country. This is far from the case. America, for instance, has phones that works on frequencies that are different from almost every country, due to the way they adapted to the GSM standard. As a result, a 'quad-band' phone is generally needed to receive calls on a phone that will work internationally.[3] (Mennen 2005)

In any case, the maximum frequency the mobile phone has been known to use is 1900 MHz. This places the phone in the radio part of the electromagnetic spectrum. (See Table 2.1.)

[3] Information regarding what type of handset is suited to each situation should be done on a personal level, by approaching local retailers and service providers.

As a result, the mobile phone has often been termed a *mobile radio* — the cellular system itself has also been called a ‘cellular radio’, but that term is receiving less use. It transmits and receives in a *wireless* environment, and as such, is part of a *wireless local area network*.

The mobile phone itself is also known as a ***mobile station*** (MS) in that is transmitting and/or receiving from a ***base station*** (BS). This BS is part of a group of cells, each with its own BS.

2.2.2 Antennas

Antennas, such as are discussed throughout the paper, are considered to be rigid and non-directional, that is, the transmitted signals radiate outwards in all directions. Effectively, the transmission and receiving happen on the one antenna. (See Chapter 4 for further details.)

These radiated signals, as shown in Section 2.2.1, are part of the electromagnetic spectrum. Therefore, these signals are also electromagnetic waves, as Section 2.4.1 further explains.

Some of the parameters that affect an antenna’s performance include resonant frequency, impedance, gain, aperture (radiation pattern), polarization, efficiency and bandwidth.

2.2.3 The Cellular Concept

The cellular concept, that makes mobile phones possible, is as simple as it is brilliant. Fig. 2.1 gives some idea of the basic concept.

A BS, or base-station (antenna) is at the center of the cell, and radiates it’s signal outwards on certain frequencies. It has adjacent to it other base-stations, also radiating out their signals on a certain frequency. This pattern of adjacent cells (base-stations radiating signals on certain frequencies) repeats all over an area, eventually covering

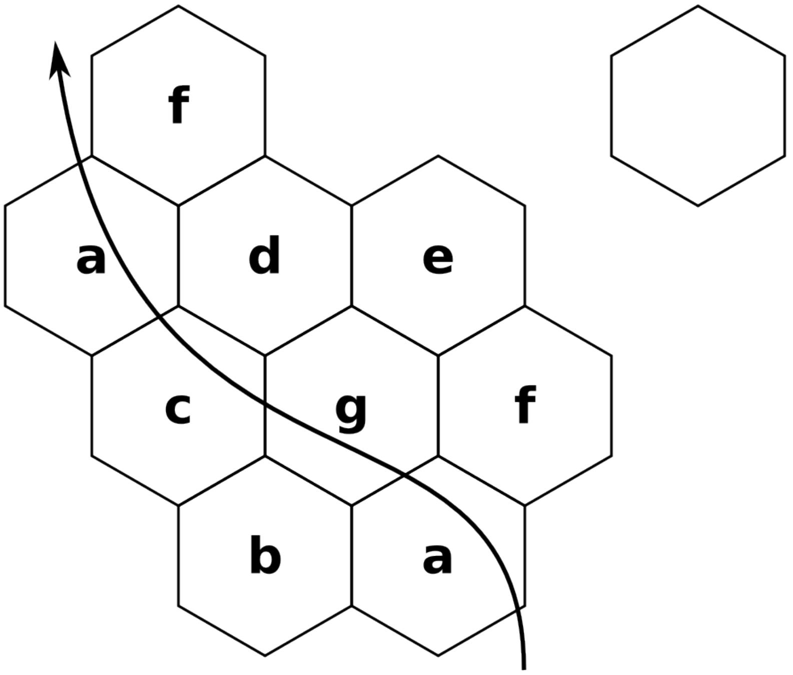

Figure 2.1: The ideal cellular concept, as conceptualised. (Agar 2003)

an entire map. (Agar 2003)

Mennen (2005) notes that base stations usually provide coverage within the range of 10 miles[4] — more than this can overstep the ACMA's "guidelines of safety limits for human exposure".[5] The usual range is often much smaller, however, as is explained in Chapter 4's section on antennas.

Fig. 2.1 shows the cellular concept as it was originally conceptualised, with a mobile (the arrow) moving through a hexagonal arrangement of cells, with each cell being neatly delimited. In reality, however, cell boundaries are not so clear, as can be seen in Fig. 2.2.

In the real world, there is a marked difference in how far the frequencies travel from the base station (BS). This is termed the ***cell radius***. This is not a fault, but a feature. Such overlap of cell frequencies allows for "***cell handover***". This is the process whereby the mobile phone stops receiving signal from the BS in one cell and

[4] 16 km, with an average number of subscribers of over 24,000

[5] The ACMA is the Australian Communications and Media Authority.

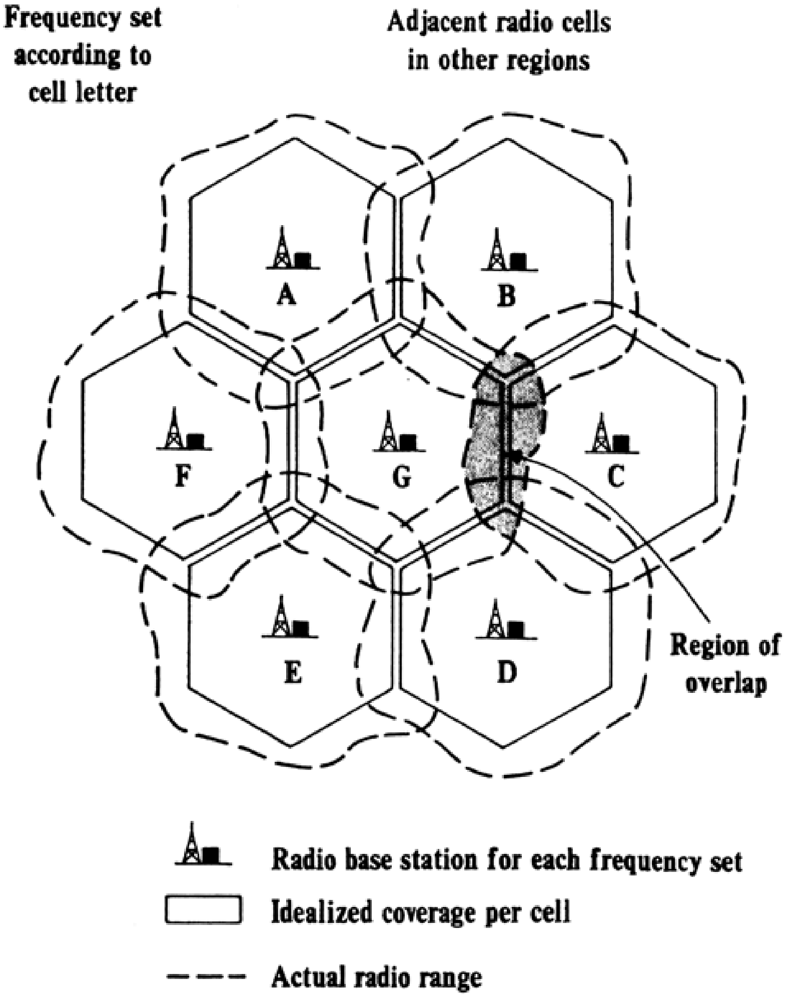
Frequency set according to cell letter
Adjacent radio cells in other regions
A
B
F
G
C
E
D
Region of overlap
Radio base station for each frequency set
Idealized coverage per cell
Actual radio range

Figure 2.2: The reality of the cellular concept. Note the frequency overlap between each cell. (Macario 1997)

shifts to receiving signals from the BS in another cell. This would only be possible if the cell radii' were overlapping.

However, such overlapping of ranges does create problems. Although it makes the cellular system possible, it also generates problems...namely the phenomena of fading.

2.3 Fading and Interference

When a transmitted signal is received, often it is not as strong as when it was sent. This is due to the phenomena of ***fading***.

Fading affects the signal in terms of amplitude, time, and phase, and it occurs while the signal is traveling through the transmission medium (generally air, in terms of the terrestrial mobile channel, but other mediums are possible, particularly when referring to a satellite link).

Amplitude is affected by means of ***attenuation*** — which has also been termed, in some circles, as 'clamping'. That is, the amplitude of the signal by the time it reaches the receiver is much reduced — clamped. This is because of a gradual loss of intensity (amplitude) in the signal as it travels through the medium.

Fading affects frequency by means of a ***delay***. As the signal travels through the medium, it may encounter obstacles. The signal 'bounces' off these obstacles, and thus arrives at a later time at the receiver than those signals that travelled by a different route — such as by Line-of-Sight (LOS) or via a different obstacle.

Finally, fading also affects the phase through a ***phase shift***. When the faded signal reaches the receiver, it is in a different phase to the original signal. This often leads to destructive/constructive interference, as per basic physics.

As fading has such an effect on the channel, there is a special section of equations and theory devoted to its study. This is known as ***multipath propagation***.

2.3.1 Multipath Propagation

It is *not* true that radio signals are easily described.

A terrestrial land-line system[6] has the advantage over the cellular system, in that its signals are solely digital, and are transmitted via cable — whose losses are known and relatively easily calculated for each type of cable used. In contrast, the mobile system is not ”pure” digital but tends more towards analog. Also, its signals are transmitted over not via cable, but over air. As a result, its losses are harder to predict, and vary depending on the situation the mobile phone is in. That is, the mobile phone is also affected by its own movement, in addition to the fading mentioned earlier. (Macario 1997)

The study of *multipath propagation* has to do with understanding these losses, and their effect on the characterization of the signal. This characterization is not so easy to predict, and thus to model.

Whenever the mobile moves relative to aerial — within the field of signal — that is, it is in relative motion to the BS, there is a ”Doppler shift of a frequency components within the received signal”[7]. (Macario 1997) Moreover, the signal undergoes amplitude, frequency, and phase shifts as mentioned in Section 2.3.

As a result, it becomes easier to *predict* the signal through probability functions than it is to calculate it exactly. There are three particular probability models that are used to determine the effect of multipath propagation on the signal: Gaussian, Rice, and Rayleigh. For the sake of brevity, these will be considered in full in Appendix B during Section B.1.

[6] The ”opposition” to the mobile phone system.

[7] This will also be explained further in Section 2.4.1,

2.4 Signals, Channels and Channel Capacity

Work into channel capacity, and its limits, was first started by Shannon, as was mentioned in the beginning of Chapter 2, and in many other research papers besides. (Ibnkahla 2005, Goldsmith et al. 2003)

Once again, though, some terminology definition.

2.4.1 Signal vs Channels

A channel is *not* a signal — a signal is *not* a signal. However, a signal can be within a channel, and yet a channel can exist without a signal. Moreover, a signal needs a channel to exist. What did all that mean?

A ***signal***, put simply, is an electromagnetic (EM) wave — that is, is a part of the electromagnetic spectrum, and thus has certain characteristics due it as EM wave. Some basic characteristics include wavelength, frequency, amplitude, phase, phase-shift, and so forth. Additional characteristics (and perhaps less well known) include harmonics, Doppler shift, and the Doppler power spectrum.

The mathematical model of the signal equation and the resulting equation are called the ***signal model***. From this model, parameters can be found to describe particular signals. For instance, a signal model for a sinusoidal signal would be:

$$x(t) = A \sin(\Omega t + \varphi) \tag{2.1}$$

and for summation of signals...

$$x(t) = A_1 \sin(\Omega_1 t + \varphi_1) + A_2 \sin(\Omega_2 t + \varphi_2) \tag{2.2}$$

where the parameters A_1 and A_2 represent amplitudes of each signal, Ω_1 and Ω_2 represent frequencies, and φ_1 and φ_2 are the phase shift of each. (Leis 2002)

At this point, it should be noted that media and the common people tend to use "channel" to refer to a band of frequencies used for media (radio or TV) transmission, aka 'radio channel', 'TV channel', et cetera. This usage tends to distort the true capability of a channel. The definition of channel that follows, therefore, is the engineering definition, as applied in signal processing.

The ***channel*** is the medium the signal exists within — that is, the "transmission medium". In terms of a cellular network, that medium is the air (as has been previously mentioned in various sections). Both the base-station and the mobile-user can be considered (relatively) close to sea-level, in terms of how the channel itself operates[8].

Modeling a channel requires, in general, knowing the ***permittivity***, that is, how well a medium *transmits* an electric field, and thus how well it will transmit an EM wave. Because the medium with a mobile channel is air, permittivity of air is generally considered to be 1.

This is a very clear distinction. A channel is not a signal. A channel may have (or transmit) a signal, but a signal may not have a channel.

2.4.2 Capacity and Shannon's Limit

Channel capacity was defined, in part, in Chapter 1, as it was also mentioned briefly in the introduction above.

Shannon's work, in general, provided a link between the maximum information in a 'communications *channel*', that is, a band of frequencies that have been used to transmit to transmit information. His work, however, has been subject to much research and development. Indeed, Goldsmith et al. (2003) notes that "there are multiple Shannon theoretic capacity definitions and, for each definition, different correlation models and channel information assumptions that we consider". The study of Shannon's work is the area often termed 'information theory', which is not to be confused with 'information technology'.

As there are so many forms of the definition, not all of them can be covered here.

[8] That is, "operates", in terms of its properties and its effects on signals that pass (are transmitted and/or received) through the medium.

However, the basic theorem can be stated as follows:

Theorem 1. *Given the bandwidth and signal-to-noise ratio, the maximum capacity of a communications channel*[9] *to carry information with no more than an arbitrary error rate can be defined.*

or

$$C \propto BSNR \tag{2.3}$$

where B is the bandwidth of the channel C with a certain signal-to-noise ratio SNR. This relationship is a logarithmic (base-10) relationship

The important thing to learn from this is the following: all antenna systems are based on this theorem, in whatever their form, be they MIMO or SISO — these terms will be explained in Chapter 4. All systems, therefore, will be limited in some way in the amount of information (voice, video, et cetera) that can be used as throughput without said information being corrupted by errors.

The amount (or rate) of errors present in the information after transmission is generally known as the ***bit error rate*** or BER.

2.5 Conclusions

In summary, there is always a foundation that needs to be laid, before one can proceed to build a house. This chapter is that foundation, and MIMO-OFDM is the house. As Macario (1997) noted,

> "Cellular radio is a complex technological system. It embraces several disciplines of engineering and has taken much enterprise and development to assemble into global systems. For example, [it] requires the combining of many large-scale technologies."

[9] That is, a system of signals being transmitted and/or received.

One of the main difficulties, in combining such diverse disciplines and technologies, involves finding ways to reconcile terms whose definitions vary across applications and disciplines. Finding definitions for such terms is not always easy, and that is only the start of the process of reconciliation. For instance, as noted above, the meaning of ‘channel’ is one such term that varies across disciplines, though it was not by any means the only one. Having a more standardised terminology (nomenclature) across all disciplines would go a long way towards increasing understanding and development of future development of ideas.

However, having given a brief cover of some of these technologies, Chapter 3 and Chapter 4 cover the theories of OFDM and MIMO respectively, with chapter Chapter 5 combining these techniques into one cohesive model, or MIMO-OFDM.

Chapter 3

OFDM

As a technique, ***OFDM***, or **O**rthogonal **F**requency **D**ivision **M**ultiplexing has found wide acceptance. It has been used for Digital Audio Broadcasting, for example, for the Asymmetric Digital Subscriber Line (ADSL), and in Europe for as the basis for Digital HD-TV broadcasting. It has even been used as the basis for the IEEE 802.11 standard for wireless LAN (WiFi).

It should be noted that various incarnations of the technique have been discussed in recent years, but what will be discussed here is the classic version, on which there is the most research.

3.1 OFDM: What Is It?

The concept of dividing up a "signal frequency band" was recognised early in signal processing. Such a concept was an appealing way to increase "system robustness against amplitude and phase distortion introduced by the communication channels, impulse noise, etc". (Ibnkahla 2005) One of the methods that were used to achieve this was Frequency Division Multiplexing (FDM).

FDM attempted to achieve the dividing up of frequency band, by sending multiple signals over a channel simultaneously.

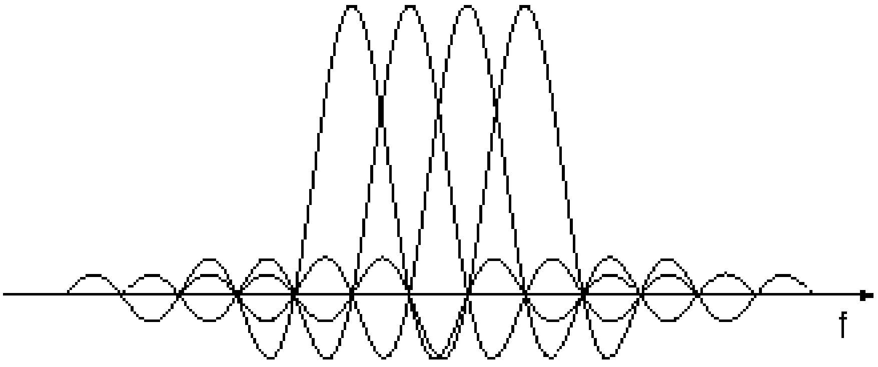

Figure 3.1: Figure depicting orthogonal sub-carriers. (Dörpinghaus & Speth 1999)

OFDM expands on the FDM method by introducing orthogonality. *Orthogonality* is the property of 'being at right angles', that is, of being separated. In OFDM, then, a single original signal is divided down into multiple sub-carriers that are separated from one another. These sub-carriers, using the techniques of FDM, can be sent simultaneously down the transmission path. (See Fig. 3.1.)

The perfect tool for this is the Fourier transform.

Without going into an involved discussion of the theorems' of the Fourier transform, these are the main things that it is important to note about the Fourier transform (not in any particular order):

- It transforms from the time domain to the frequency domain.
  - The *inverse* Fourier transform returns the function from the frequency domain to the time domain.
- Applying the transform introduces complex and real terms that are exponentials.
- It is based on Euler's formula.
- It separates the "harmonics" in the signal.

In terms of linguistics, the phrase "Fourier transform" has come to apply equally to the frequency domain representation of a function as it does to the process that transforms that function from one domain to another.

Interestingly, the Fourier transform also has a close relationship to sampling a signal.

3.2 Sampling a Signal

A knowledge of sampling — measuring a signal at regular intervals in order to convert into digital form — and the process of signal reconstruction from a sampled signal (obtaining the original waveform) is important in OFDM. OFDM is a signal modification technique, and as such, it has explicitly (and implicitly) to do with sampling and signal construction.

When sampling a signal, one wants to sample it often enough times to avoid ***aliasing***, or errors in sampling that have made the sampled signal appear than it truly is. This is due to the *resolution* of the sample — how often it was sampled, or the sampling frequency.

Applying the Fourier transform (and its inverse) to a transmitting signal, it is possible to show that an incoming signal $m(t)$, sampled at a uniform interval T_s, can be represented by

$$m(t) = \sum_{n=-\infty}^{\infty} m(nT_s) \frac{\sin \omega_M (t - nT_s)}{\omega_M (t - nT_s)} \tag{3.1}$$

where ω_M, the upper limit of the frequency domain of the signal $M(\omega)$, is $\omega_M = n/T_s$ and n is the n^{th} sample. This Eqn. (3.2) is the *Nyquist-Shannon interpolation formula*, and is also sometimes called the cardinal series. (Gibson 1999)

An extension of the above is that f_s, the minimum sampling frequency, is such that $1/T_s = f_s$, which is also known as the ***Nyquist rate***. The Nyquist rate — or, twice the highest frequency present in the signal — is the the minimum rate at a signal can be sampled without introducing errors into that sample. (Leis 2002)

As a result of this Fourier process however, the resultant signal will often require a *guard interval*. This is due to the ***sampling theorem***.

3.2.1 The Sampling Theorem

The sampling theorem can be stated as follows.

Theorem 2 (Sampling Theorem from Leis (2002))**.** *The minimum sampling frequency must be at least twice that of the highest frequency component present in the original signal.*

This is one of the more fundamental statements of signal processing and, indeed, most engineering fields related to communications.

Using the signal from Section 3.2, $m(t)$ sampled at uniform interval T_s, this sampling interval T_s is known to be related to the maximum sampling frequency f_M by

$$T_s = 1/(2f_M) \tag{3.2}$$

This relationship is called the ***Nyquist interval***. Therefore, the Nyquist rate extends itself such that

$$f_s = 1/T_s = 2f_M \tag{3.3}$$

When $f_s < 2f_M$, that is, it is less than the Nyquist rate, that is known as *aliasing*, or *foldover*. This makes the signal distorted and it becomes "impossible to recover $m(t)$ from the sampled signal". (Gibson 1999)

When $f_s > 2f_M$, that is, it is greater than the Nyquist rate, there is a gap, known as the guard band, or guard interval.

Hence, sampling a signal at a sample rate higher than the Nyquist rate makes it easier to recover to the original signal from the sampled signal. This process is known as signal reconstruction.

With knowledge of the above, it is possible to build a working model for the OFDM technique.

3.3 OFDM: The Model

The mathematical model is...

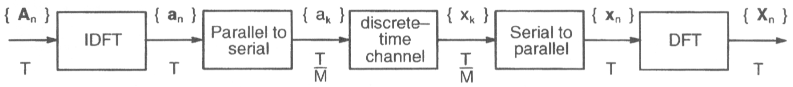

Figure 3.2: A block diagram of the OFDM system model. (Gibson 1999)

$$z_{l,k} = a_{l,k}H_{l,k} + n \tag{3.4}$$

where...

z is the output of the channel

a is the tap weights

H is the transfer function at

l time-slot and

k sub-carrier,

n is noise (AWGN)

In particular, H is of most interest, as it represents the OFDM channel modulation technique. In particular,

$$H[l,k] = \begin{bmatrix} H_{1,1}[l,k] & H_{1,2}[l,k] & \dots & H_{1,K}[l,k] \\ H_{2,1}[l,k] & H_{2,2}[l,k] & \dots & H_{2,K}[l,k] \\ \vdots & \vdots & \ddots & \vdots \\ H_{L,1}[l,k] & H_{L,2}[l,k] & \dots & H_{L,K}[l,k] \end{bmatrix} [] \tag{3.5}$$

provides the $L \times K$ matrix corresponding to the l^{th} sub-carrier and k^{th} OFDM symbol.

Perhaps a simpler way of understanding it would be in Fig. 3.2

It breaks the signal down into sub-carriers, modulates them using the IFFT and sends these through the channel. These are picked up at the receiver. It is then up to the receiver to use the FFT to reconstruct the sub-carriers (demodulate) back into the original signal.

3.4 Conclusions

This, therefore, is OFDM. It is a way of modulating and demodulating the signal, to send it through the channel, by means of multiplexing — breaking it up into portions — through use of the IFFT and FFT. In Chapter 5, it will be proved how MIMO and OFDM can be combined to form the cohesive theory of MIMO-OFDM.

Chapter 4

MIMO

be everywhere. All such forms mentioned receive a signal — but only the first pair can truly receive *and* transmit. Specifically, it is the mobile phone "towers", or antennas, that will be the subject of this dissertation.

In recent decades, the antenna has increasingly become a more common sight on modern skylines in all its forms. However, the particular antenna considered suitable for a cellular system is only one form — one sub-species, perhaps — of this type of device. This chapter will consider an extension to the antenna: the antenna array, or as its application is increasingly becoming known, MIMO.

4.1 MIMO: What Is It?

Multiple antenna systems have been a source of increasing research and discussion over the last decade or so, as interest in the MIMO-OFDM technique has grown and papers have been published. One cannot hope to reproduce all such data, or material, but could, perhaps, give some sort of overview instead.

The ***MIMO*** technique itself, as might have been mentioned earlier, is an acronym that stands for **M**ultiple **I**nput **M**ultiple **O**utput. It is a "step up", then, from the stand antenna system that has been commonly in use for decades (as at the time of writing), which is commonly known as SISO.

SISO stands for **S**ingle **I**nput **S**ingle **O**utput. It is the single antenna for a single cell, with a single user — one antenna, using a single frequency range — servicing a single antenna on each mobile phone. This is by far the most common system in place (for the cellular system) at the time of writing.

Both MIMO and SISO refer to antenna systems and their relationships to the user, the mobile phone (and the mobile phone's antenna).

4.2 The SISO Problem

The antenna itself, as described in Chapter 2, was assumed to be a rigid and non-directional device in most cases. In that chapter, it was also mentioned that the transmitter and receiver are assumed to operate on the one antenna. Fig. 4.1 shows how this is possible. Indeed, this combination unit antenna is what makes possible the cellular system.

This was stated simply for ease of conceptualisation. Indeed, the initial antennas that are placed in a cell system as base-stations are omni-directional. due to the fact that the goal is to achieve coverage quickly. But as Hernando & Pérez-Fontán (1999) notes, this is not true for mature cell systems.

> When the network has been in operation for a given period, directive antennas are more suitable to improve the carrier-to-interference ratio and thus reduce the cochannel reuse distance to increase network capacity. In these mature cellular networks the antenna directivity defines 120-degree cells. Sometimes six-sector cells (60 degrees) may be used.

As Hernando & Pérez-Fontán (1999) also goes on to point out, such a 'immature' system has the risk of being able to provide for its users.

Using the communications terminology for 'channels' to refer to a band of frequencies,

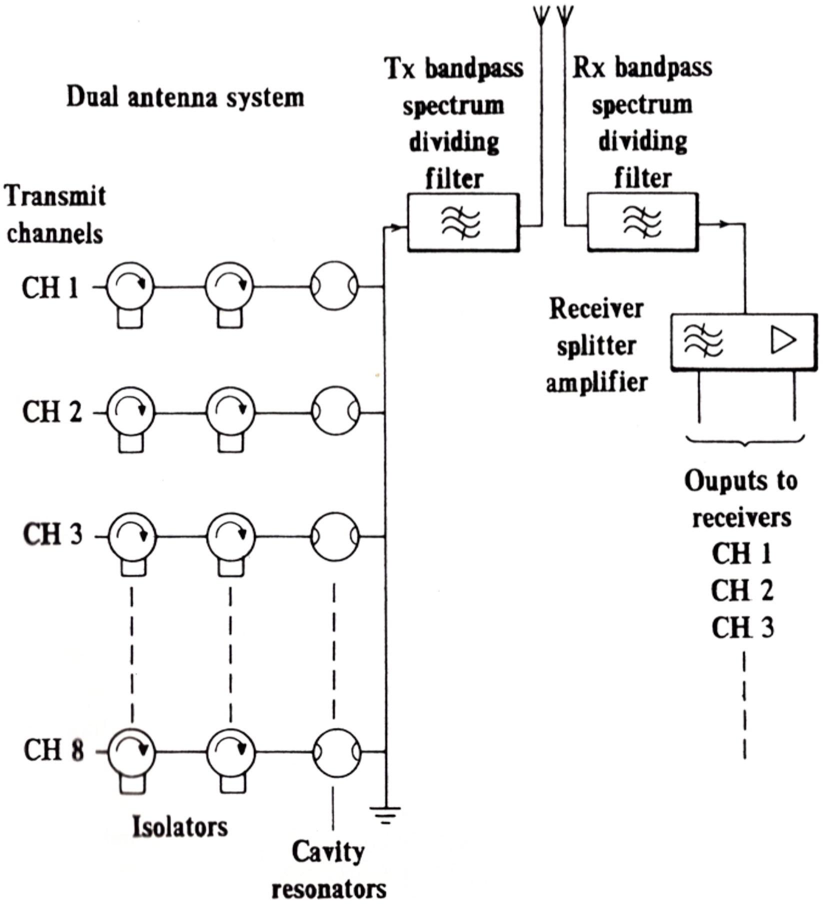

Figure 4.1: Depiction of the combination unit antenna — a transmitter and receiver operating on the one antenna. (Macario 1997)

Hernando & Pérez-Fontán (1999) gave the Erlang-B formula,

$$\begin{aligned} p_t &= B(A, N) \\ &= \frac{\frac{A^N}{N!}}{\sum_{k=0}^{N} \frac{A^k}{k!}} \end{aligned} \tag{4.1}$$

to describe the probability (p_t) of unsuccessful call attempts, where A is the offered traffic and N is the number of traffic 'channels'. This Eqn. (4.2) allows for a way to predict to the number of 'channels' required by a base-station for a certain GoS (Guarantee of Service).[1]

Using this Erlang-B formula, Hernando & Pérez-Fontán (1999) proved that an antenna of radius 10 km, with an assumed probability that 2% of calls will not get through (p_t of 2%) will require 328 channels — an excessive, if not impossible amount. However, if the radius is reduced 1.5 km, and assuming uniform traffic, with the same blocking probabilities, the amount of channels required would only be 13. This is much more within the realm of reality.

However, there are still problems with the cellular system. Frequencies can not be reused too close together, and interchannel (or cochannel) interference is still a big problem, as well as other problems facing the cellular system. It is hoped the MIMO will be a solution.

4.2.1 The Antenna Array

The ***antenna array*** is a term that describes a number of antennas in a group, that also radiate and receive signals (on frequency bands) as a group. The key factor is that these antennas are no longer omni-direction (all directions) but directional.

At this point, it should be noted that it is not the purpose of the dissertation to criticise current or suggested models of antenna arrays. Nor is this paper a design document.

In conducting research, it was found that many of the available printed texts focused on the GSM model, as this has been the model most widely implemented, although

[1] GoS in percentage units is given by: $100[1-(1-p_t)\cdot p_c]$ where p_c is the typical coverage probability.

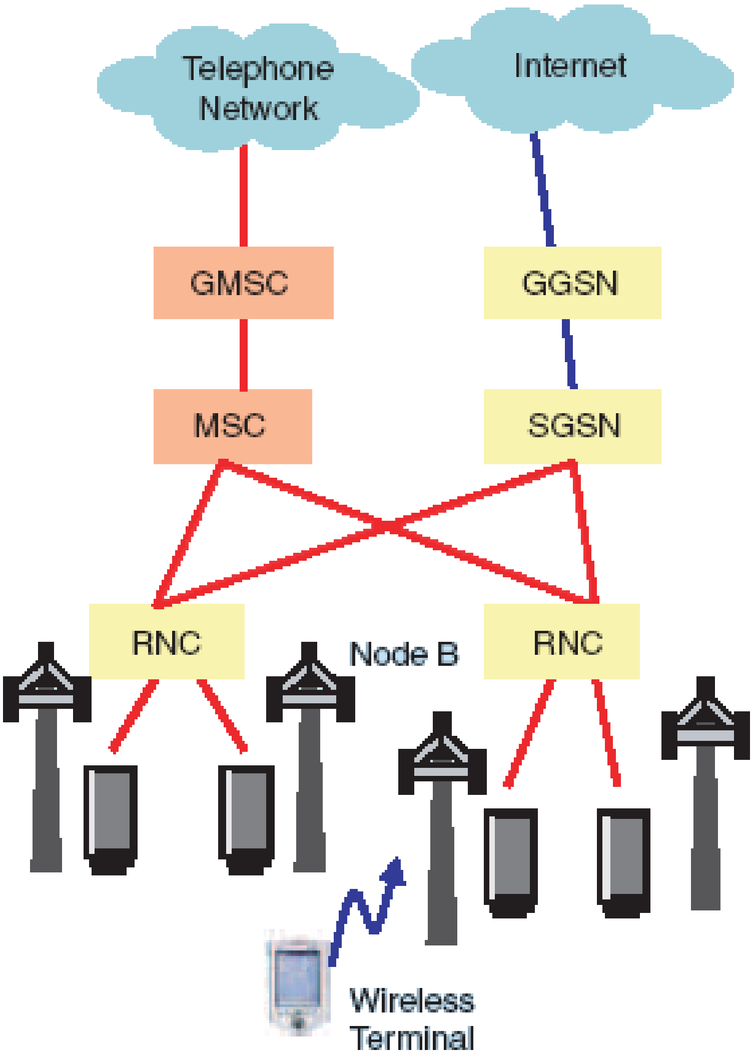

Figure 4.2: Model of a GSM system.(Laroia et al. 2004)

in various forms and in varying degrees of success. In general, the GSM model is implemented as shown in Fig. 4.2. In such a system, the interface between the BS and the networks (telephone or internet) is the bottleneck, and could be considered to be the cause of many problems.

This GSM model is made of up of interconnected disparate pieces. As noted by Laroia et al. (2004),

> A radio network controller (RNC) controls several Node Bs. An RNC directs the voice traffic to the mobile switching center (MSC) and the data traffic to the serving general packet radio services (GPRS) support node (SGSN). The MSC is connected to public switched telephone network (PSTN) through the gateway mobile switching center (GMSC). The SGSN is connected to the Internet through the gateway GPRS support node (GGSN).

The MIMO array solves many of these problems, even though the model itself is still under development, and is therefore subject to theoretic change with each new paper that is published. At its heart, however, seem to be a few basic concepts and principles, which will be presented in Section 4.2.2

4.2.2 MIMO: The Model

The mathematical model of MIMO is as follows:

$$Y = HX + E \tag{4.2}$$

where

$$
\begin{aligned}
Y &= \text{outputsignal}\\
H &= \text{channelmatrix}\\
X &= \text{inputsignal}\\
E &= \text{thermalnoise}
\end{aligned}
$$

Being a multiple array, all these terms. Had it been a single-antenna-to-single-antenna (SISO) system, all these terms would have been scalar.

Please note, however, that finding an appropriate diagram of the MIMO system was not as easy as it seemed. It changed with every paper that discussed, and there seemed to be no real 'standard' design available. Even the equation in Eqn. (4.2.2) is a conglomeration of equations from various research papers, and so cannot be cited.

4.3 Conclusions

This, therefore, is the definition of MIMO that is applied within this paper. When "multiple input multiple output", or MIMO, is mentioned. It refers to a base station (BS) of multiple antennas, placed in an array, transmitting on a frequency range, to service a mobile phone, also with multiple antennas placed in array. In Chapter 5, it will be proved how MIMO and OFDM can be combined to form the cohesive theory of MIMO-OFDM.

Chapter 5

MIMO-OFDM: The Concept

It might have been noticed that there is, to some extent, some repetition of topics throughout chapters Chapter 3 and Chapter 4. This, unfortunately, had to be done to explain the full breadth of each topic, although it was kept to a minimum to avoid "re-inventing the wheel". In this chapter, however, these topics will be combined, to show how MIMO-OFDM actually *works*.

5.1 MIMO and OFDM Together

Fig. 5.1 shows a standard (concept) view of how MIMO and OFDM may be incorporated to make one complete system.

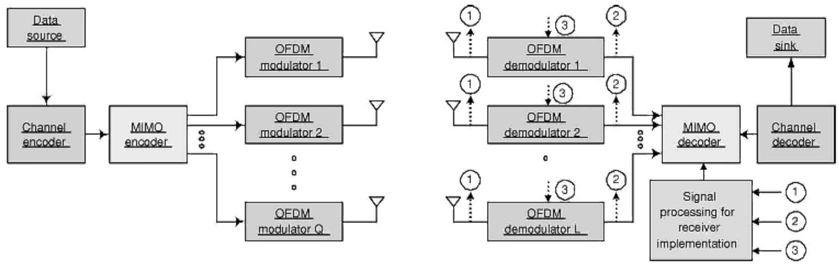

Figure 5.1: The standard model of the MIMO-OFDM system model, where Q and L are the numbers of inputs and outputs. (Barry et al. 2004)

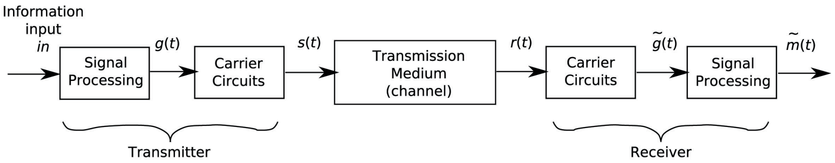

Figure 5.2: The standard model of an antenna system.

Note that the MIMO array leads directly into the signal processing, the OFDM. Compare this to the GSM model shown in Fig. 4.2, and the antenna model shown in Fig. 4.1 in Chapter 4, as well as the block diagram shown in Fig. 5.2.

Simply put, MIMO-OFDM *stream-lines* the antenna system. It removes the 'bottlenecks' and allows the antenna almost direct access to the channel.

In addition, within OFDM, the technique itself offsets interference.

One of the main problems with a cellular system, as was discussed in Chapter 2, was multipath propagation. (Full details in Appendix B). The methods that OFDM applies to break-down (sample) and then reconstruct a channel using the IFFT and FFT are actually based on how a multipath-affected signal works, not against it. There, there is a reduction in the effect of noise at both the transmitter and the receiver.

Moreover, with careful placement of the MIMO antenna array, signal can be aimed in certain directions. This will reduce, or exaggerate, the impact of the phenomena of ***beamforming*** as desired. Beamforming is where a directional antenna array's signal is concentrated in one area. With careful planning and placement of the MIMO array, this can either become a detriment, or a feature.

5.2 Drawbacks

It would not be fair, however, to present a glowing report without at least considering the disadvantages.

For the most part, this is what the simulation was designed to discover, and for this, one should consider reading Chapter 6 to Chapter 7, or Chapter 8 for the overall conclusions.

Also, there were also limits, to how much one could discover through research alone, mainly due to problems with notation varying between disciplines. As there has been so much to cover in the proceeding chapters, this has researching in a wide range fields, and not always in the one area of "signal processing". This has had consequences, particularly in notation, and in the number of different papers that had to be read. In particular, notation and terminology has not been static. It has changed depending on the field and the topic being looked at that particular moment.

Perhaps, just like there is a standardised system of units (the SI units), there could be a standardised system of notation. It would be, if nothing else, interesting to implement. (See Chapter 8 for details.)

5.3 Conclusion

MIMO-OFDM is something that truly needs to be implemented to discover its advantages and disadvantages. Although there were problems with researching, due to notation and concept changes across various texts, there is enough confidence in the basic concept to continue with the simulation. This will be discussed in Chapter 6 and its results presented and analysed in Chapter 7.

Chapter 6

Simulation

The specification of the project was clear. The simulation had to be in C++/C, or MATLAB. The decision was made early to avoid MATLAB as much as was possible (being a vendor-locked programming language). The reasons for this lie in the capabilities of each language — and will not be debated in full here.

However, one of the main reasons for the choice was the ability to fork. Research indicated that whatever direction the final solution went, it had to have the capability to maximise the use of the cores of an SMP (**S**ymmetric **M**utli**P**rocessing)[1], which what are the common processor model common today.

However, ability to fork not one of MATLAB's outstanding features. However, C++ does fork, and does it well. Moreover, creating a program in C++ allows for platform inter-operability, and a personal goal of the project was to create a simulation that would work in both Windows and Linux. The differences between the two OS's are actually rather small, as long as one as holds true to the C++ ***STL***, or Standard Template Libary.

This language was therefore chosen as the main simulation programming language, as well as for its memory handling features and advanced containers[2].

[1] An architecture where additional cores share memory, and are added as volume of processes increases.

[2] A container in C++ is a holder object that stores inside it a collection of other objects, or data,

As such, when terms are capitalised (such as Signal instead of signal), it generally refers to that term as used in the simulation. Moreover, the same term capatised but as `Signal` typeface, refers specifically to the way it was implemented in the simulation.

6.1 Programming in C++

There is one thing that should be absolutely clear, before we go much further.

Programming a complicated series of equations, particularly mathematical and scientific equations, is not as easy as it sounds. Whatever the language.

C++, in particular, cannot represent equations directly from the printed page to the language notation. The main reason for this is the ˆ (caret), which it uses as a bitwise operator. The power operations have to be coded as separate functions — increasing program complexity. MATLAB, as mentioned earlier, is not the best, as while it uses the ˆ as a power operator, it is computationally intensive. It also does not guarantee results — at least not without paying extra for “Toolboxes”.

That being said, test runs *were* carried out, in both MATLAB and C++, of some generic signal-processing equations. The results were...not pretty. Not only were the computations to obtain a single matrix of results relatively intensive, but the results were far from what was being hoped to achieve.

MATLAB produced results, but they were far from expected. C++ did not even manage to compile.

Research into MIMO and OFDM continued, however, in the hopes that a solution would be found. It did some months of battling with the languages, but in the end there was a break-though. Unfortunately, this was also late in to the project (approximately around the beginning of September). The rushed timeframe meant that, unfortunately, limited external documentation could be produced.

and provides with itself a number of additional functions.

6.1.1 The Light In The Tunnel

The solution, such as it was, was simple.

A computer itself is a binary machine. It represents numbers (the decimals we humans are familiar with) and expresses them as binaries (on a base-2 system). As such, it uses *boolean* (true or false) logic, much like DC electrical signals, which are often stated as 1 or 0, on or off. A binary number is actually just a collection of bits — singular values (or a quantum) that make the whole (the binary word).

Moreover, in terms of how signals are transmitted through air, it is not the signal itself that is being transmitted, but in truth, it is *more like the molecules in the air that are passing the signals' energy along*, much like they pass thermal (heat) energy along by vibrating. It is these molecules (the ”quantum” of the air system) that make up the whole (the air).

Combine the two thoughts, and there is the principle behind the simulation: the means of transmitting the signal through the air (the molecules) can be 'simulated' by the bits in a computer program. In other words, the construction of the simulation came from a consideration of first principles: from basic chemical and physical properties of how signals are transmitted.

This leads to the two so-called “backbones” to the simulation system: the bitstream system, as implemented by the 'signal's, and the “MACS” layer. These were combined into the final implementation, which will be discussed in detail in Section 6.2.

6.2 The Final Implementation

For future reference, Appendix D lists the code in full detail, with all modules at the time of writing, and Appendix C gives sample runs with current (at time of writing) attempts of producing output.[3]

[3] At some point, producing the dissertation document had to take precendence over code fine-tuning and output production, particularly due to the rapid developed and shortened code development cycle.

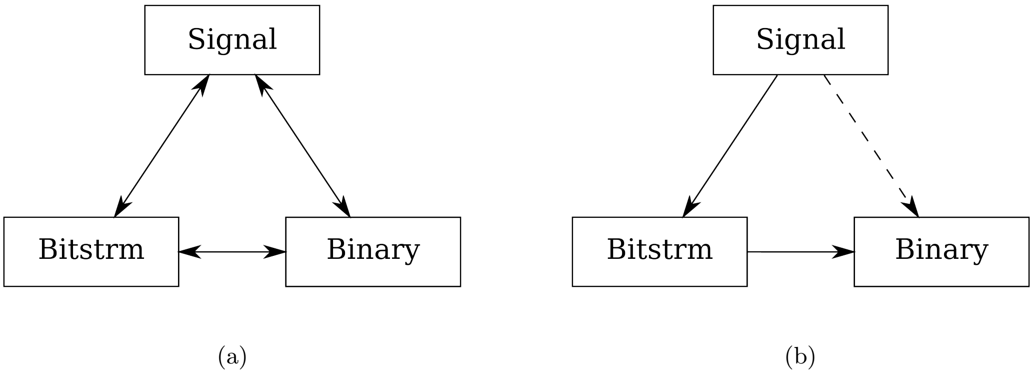

Figure 6.1: Flowcharts depicting the flow of (6.1a) data and (6.1b) execution between a signal and its associated data: a bit stream and a binary.

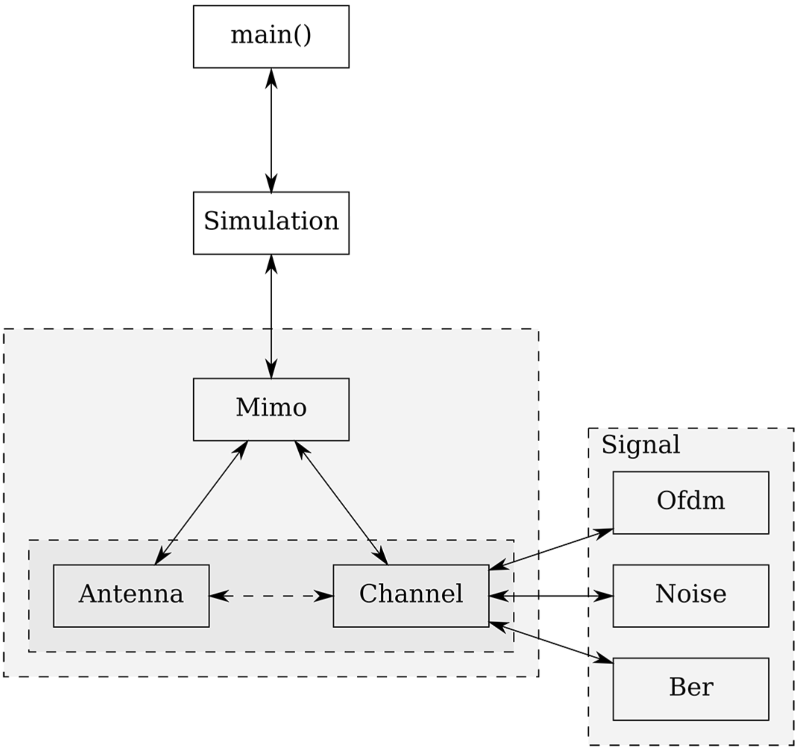

Figure 6.2: Flowchart the overall program flow and the connection between the classes.

Fig. 6.1 provides the flow-charts depicting the implementation of signals and the bit "streams", and how these interacted with the custom-made Binary. Data was shared among them fairly equally, and execution flowed neatly from `Signal` to `Binary`.

In Fig. 6.2, each of the `Ofdm`, `Noise` and `Ber` are all objects of type `Signal` as shown — they are all implicitly interacting with the '`BitsStrm`' and `Binary`' model shown in Fig. 6.1. Moreover, the dashed box around the `Mimo`, `Antenna` and `Channel` types show how the MACS system was implemented. Each of these share standard values that can be equally and easily accessed. This is through a separately defined system, though not shown in Fig. 6.2 for brevity, its listing is in Section D.2 of Appendix D.

6.2.1 Binaries, Bits, and Streams

As mentioned in Section 6.1.1, the principle aim is to simulate signal transmission using the inherent units of a computer: the bits.

This was both easier, and harder, than it seemed.

It was only when the program (current version) was nearing completion that the <bitset> header was found, which is a C/C++ standard header that implements much of what was trying to be achieved here. It probably would have saved much work, and debugging, even though it did not quite fit the specifications. However, as it was found so late in the programming cycle, it could not be included.

The decision was made not to rewrite but to continue with 'non-standard' custom files and implementations, which will be described as follows.

A Binary System: In Brief

The one problem with using a "binary" system is that it varies. There is no *one* standard, just as the decimal system also has scientific notation (which itself can be represented as $\times\ 10^x$ or $1Ex$). Each architecture has different versions and implementations of the system.

There were two options:

1. catering for each architechture

2. implementing a custom binary system

The other challenge was binary word-size. With the increasing advent of 64-bit operating systems (to take advantage of the SMP architecture), the values of previously standard sizes in C/C++ (such as `int`, `char`, et cetera) have changed depending on particular computer of compilation and execution, and cannot be guaranteed for 64-bit suitability. Along similar lines, Windows has added the data type of `wchar_t` to their programs for their "expanded unicode" character set, which is specifically for the Windows operating system, and is only supported on that system — and by Microsoft specific compilers. It was not used here, as it was known in advance that the program would be tested on the Linux OS. Instead, the standard ASCII character set was used.

Allowing for 64-bit values and platform-interopability was important, as this simulation involved a lot of number crunching. The decision was made to switch from `int`s and `char`'s to the standard size-type fittingly known as `size_t`. This data type is defined according to the maximum size that can be allocated into a data variable — which made it suitable for both 32-bit and 64-bit architectures, and any platform that implements the STL.

The solution, it seemed, was to not base on a particular computer's architecture per se, but on the binary system definition itself. This would avoid tying the program to a particular system type (Intel vs AMD) or operating system (Linux vs Windows) — apart from the use of the standard `size_t` type.

The best way is, more often in programming, the easiest way.

Given the shortened development cycle, this was the approach chosen. This is the 'Binary' module (listed in Appendix D as Section D.15) that stores both a decimal (base-10) number and its binary equivalent (base-2), and has all the standard operators (+, -, ×, et cetera). It also has the extra capability of giving a C++ style string on request.

6.2.2 Streaming Data

Proponents of C++ talk quite often about “streams”, “file streams”, “input/output streams”, and even “overloading streams”. To the unwary, and newcomers, such terms sound disturbing. Streams in C++ are simply a means passing data too, and from, objects, or places — such as file, or wherever I/O is taking place. This is often achieved using the operators `<<` and `>>` a bit like `+` and `-` — such as ‘adding’ values to an output ’stream’ (location) and ’taking’ values from some input ’stream’ (location).

Such streams get their names from the often long lines of `>>` and `<<` operators, i.e.

```
    cout << " tmp = " << tmp << " \& count " << count << endl;
```

where `cout` was the output stream (in C++, it is assumed to be to the console or terminal) and `tmp` and `count` were some variables whose variables are to be displayed. If `tmp` and `count` had values of, say, 4 and 9, respectively, then such a line would produce at the console:

```
    tmp = 4 and count = 9
```

Although, in this program, there was oftem mention of a bit-stream (it was actually termed a `BitsStrm` to avoid confusion with other C++ classes and basic types), it did not, in actual fact, operate as the above mentioned “streams” in the sense of having the ability to use the `<<` and `>>` operators. These two operators are actually defined originally in C (of which C++ is a derivative language) as *bit* operators. To ‘overload’ them and convert them into i/o operators would not be a wise course in this case as it being a detractor from the nature from the nature of the container.

It was refered to as a “stream”, however, as it was a means of moving (passing) bits easily from one place to another. In this sense, it fullfilled the definition of a stream. See Section 6.2 or Appendix D for futher details of how this was accomplished.

6.2.3 Streams as Signals

As was mentioned in Section 6.1.1, the idea behind the simulation was to represent each Signal in terms of their physical representation; on the computer, this meant using a ”stream” of bits.

There is, at this point in time, no C++ ability to handle such capability — even though “streaming” data from one point to another is a basic part of its functionality. However, it does come with basic and high-level ‘containers’. For instance, a ‘basic’ container would be the `vector`, which is considered to be an improvement on the array (matrix), as it has better memory management. It does, however, have linear access times, due to its implementation methods.

The ‘higher-level container used instead was the ’***deque***’, as it has the advantage of constant access times to all memory, and handles its own memory.

However, one final refinement was necessary before the final implementation of Section 6.2.

MACS a.k.a MOCA and MACON

MACS stands for Multiple Antennas with Channels and Signals. Other versions of the acronym were MOCA (MIMO-OFDM with Channels in Antennas)

MOCA stands for **M**IMO-**O**FDM-**C**hannel-**A**rray system, as implemented in the program. Interestingly enough, other research papers have referred to this line of thought with similar acronyms — apparently with MAC (Multiple-Array [with] Channel) for short. According to Goldsmith et al. (2003), it has been an area of increasing research. In this text, however, due to certain methods of implementation, MAC has been also referred to as MOCA, and ***MACON*** (Multple Antenna Channl [with] Ofdm and Noise’ system). These terms (MOCA, MACON, and MAC) can be used rather inter-changeably, as long as the following is kept in mind: they refer to the interaction of antennas and channels with the MIMO-OFDM theory.

This is a critical leap in thought.

In terms of the programming language, this means that the Antennas' (capitalization deliberate) can share with the Channels (also with deliberate capitalization), and these, in turn, have something to share with the Signals (capitalization mine). What is it they share?

Their *effect* on molecules.

Moreover, continuing with the definition used throughout where a channel is what a signal travels through, and is not another way of refering to a signal or a frequency band, this drastically simplified the implementation. It meant that the Channel, instead of being another signal, could itself be used to hold signals, that is, as a way of containing the Signals implemented within the simulation.

6.3 Further Thought

Having said all that, there remains further work to be done.

Current estimates are that it would have only taken about one week to a month (depending on constraints) to produce actual data in a graphical form. The program remains, unfortunately, in a state of needing repair as it being debugged at the time of writing. (For instance, one such bug is that the OFDM signal is strangely not being fully allocated to it's proper length while all other signals are, even though it uses the same code.) Moreover, the choice of available compilers (for a 64bit XP system) will increase next year, providing more choices (that will also adhere more closely to the ISO and/or ANSI standard than the one that was used).

The obtaining of actual data and converting to graphical output (such as the coding of a program to create a svg-image using the WW3 standard) could be the subject of later research, or, optionally, private work on the part of the researcher.

Also, as mentioned in Section 6.2.1, C++ does come with the <bitset> header file, with standard binary implementations. It would be an interesting personal challenge to incorporate this into the program (perhaps in place of the binary module, listed in Appendix D, Section D.15).

6.3.1 Impressions and Recommendations

For only having been in development since September, the program implementation was able to reach a relatively advanced stage of completion. Various runs were able to be completed, with antenna array sizes up to 1024 on both the receiver and the transmitter end — although this did take a while to initialise, let alone run. To be honest, it would have been nice to have more time available to developing and play around with the environment, and taking the program further. Unfortunately, 2 months was the time frame available when the change was made from direct equation-to-language implementation to an implementation that highlighted the language features. Had the realisation been made earlier, instead of focusing so heavily on research, more could have been accomplished in the actual simulation.

Chapter 7

Simulation Results

As has been mentioned in Chapter 6, the simulation could not be implemented as initially conceived, simply because the engineering equations did not translate directly into the target language. This led to a complete change in thought, and a completely different conceptualisation of how MIMO-OFDM can be implemented as a simulation.

Unfortunately, it also meant a highly shortened development (2 months) from conception to scheduled production of the actual simulation application.

7.1 Ouptut

Due to the abbreviated development schedule, not all of the target goals were reached.

First and foremost, the OFDM component was not implemented as one would like. The signal is not broken down into sub-channels and sub-carriers and then reconstructed from such. This would require more code, adding more modules to the code, and there simply was not the time to do that and submit a dissertation as well. That said, the OFDM component had a (sampled) signal inside it, waiting to be broken down and reconstructed.

Therefore, all output of the simulation would necessarily have to be adjusted accordingly at this point in time. An estimate of how long it would take to implement the OFDM component can not be given, as the added code would have itself to be completely integrated and then debugged.

Moreover, at the time of submission, the collating of the BER component into actual output had not been completely implemented.

However, in programming, what takes the most time is the initialisation phase and debugging involved therein. One has to ensure that all the objects are initialised, that all the data is there were it should be, before one can actually proceed to *use* the data. Otherwise, using uninitialised data causes a Segmentation Fault ("SEGSEV" in programming speak), which literally means the program is trying to access empty memory, or memory it is not allowed to. This causes program's to hang, or exit inappropriately.

This is the stage the program has just completed. In two short months, it has moved from initialisation, to being able to use/output the available data. In the programming world, this is an important step.

The final debugging process ran the code through both VC++ (Windows compiler) and G++ (standard Linux compiler) to ensure platform inter-operability. It compiled and ran on both compilers, although with limited success on the VC compiler due to lack of compliance issues with standards.[1].

As such, although there is no actual technical output as such, proof of compilation without errors can be found in Appendix C, with directions for running in Section C.2.1. Plans for output are also included in Section C.4 of Appendix C. There is also a listing of all source code in Appendix D with further explanations.

[1] See Appendix C for full details.

7.2 Performance Analysis

A performance analysis, particularly in software terms, is where the program is ran through a separate program that tests it for memory leakage, unusual function calls, stack usage, et cetera. In other words, how efficient the program is, and how well it is implemented. It is a way of testing how well the program has protected itself against the "creative user" who is millions of ways to break code. If a program does not error check and is not efficiently implemented, then such programs will "break" during the performance analysis.

Specific programs to carry out this analysis can be up to and above $2,000US, far out of range of a researcher's budget constraints. As such, the decision was made to use a Linux tool, as the program compiled on Linux as well as Windows. The program was the `strace` facility, and is a standard package that can be downloaded for Linux via aptitude or synaptic, depending on the distribution.

It confirmed what was expected. There were no problems. However, due to size of the file (running unaltered, the program generated a 125kB file), it could not be included in a standard student dissertation as an appendix.[2]

7.3 Conclusions

The results and analysis that have been completed have confirmed that MIMO-OFDM certainly holds much promise.

Simulating it, when one held true to the demands of the languages involved, was actually a relatively easy matter. Although this simulation is not complete at the time of submission, running available trace programs on the simulation shows the implementation is efficient. Moreover, the implementation is capable of handling high-end arrays, that is, arrays of size 1024 × 1024. This might be greater than what current technology can do, but the simulation shows that is yet *possible.*

[2] It is, however, in the attached CD, if it is included, along with the actual program for the curious.

What is life, without hope?

Of course, this is a software implementation, not a hardware implementation. It remains to be seen if such lofty software goals can be transferred to the real world.

Chapter 8

Conclusions

8.1 Areas For Improvement

In researching this dissertation, one of the more outstanding areas of frustration had to be terminology, and the related definitions. As mobile (cellular) communications covers such a wide field, with papers being published by authors in many lands, notation changed with each paper and text along with terminology, and in some cases, within books. Each time the notation changed, the text would have to be scoured to find the latest definition of what the author was referring to, and this drastically slowed down the process.

Apparent errors in such notation could not be confirmed, as the notation was constantly changing.

There were two options:

- Try to research everything.
- Simulate from first principles.

There was, of course, not the time to try the former. Attempting that would take far more time than the dissertation time-frame allowed. The only option possible, and the one chosen, was the first. With every paper read, notation changed, the basic *idea* changed, and consistency was soon lost. The only solution was to fall back on general knowledge as an engineer, of standard chemical and physical principles as well as knowledge of how a program works, in order to construct the simulation.

This would not have had to be done if the notation and terminology had be consistent, and well explained, across the board. As this was the largest area of frustration, and caused the most change within the simulation, it needs to be highlighted first and foremost.

8.2 Final Thoughts

The final thought, and the second conclusion, is as follows.

Once terminology is actually understood, and the basic concepts are understood, MIMO and OFDM are relatively ingenious concepts. Adding antennas to increase information carrying capacity, and improving signal modulation, these are the ways of the future. This, perhaps, is what these two terms should represent. Adding definitions to the terms, and using them to the 'extra's that come with their application, clouds the issue, and should always be avoided.

That being said, simulating pure MIMO and OFDM was simple, and easy. The program that resulted, although not complete at the time of writing, has already proved itself to be efficient. Moreover, the simulation has proved that implementations of MIMO-OFDM beyond what current technology is capable of *is possible* and, eventually, achievable.

In summary, MIMO-OFDM holds much promise. It is a concept to be followed with interest by engineers, whatever their discipline.

List of References

ACMA (2009), *Electromagnetic Radiation (EMR) Safety & You Fact Sheets*, Australian Communications and Media Authority, Commonwealth of Australia.
http://www.acma.gov.au/WEB/STANDARD/pc=PC_310377
current April 2009.

Agar, J. (2003), *Constant Touch: A Global History of the Mobile Phone*, Revolutions In Science, Icon Books UK, Grange Road, Duxford, Cambridge CB3 4QF, United Kingdom.

ARSPNA (2008), *Radiation and Health Fact Sheets*, Australian Radiation Protection and Nuclear Safety Agency, Commonwealth of Australia.
http://www.arpansa.gov.au/eme/index.cfm
current April 2009.

Barry, J., McLaughlin, S., Ingram, M., Ye, L., Stüber, G. & Pratt, T. (2004), 'Broadband mimo-ofdm wireless communications', *Proceedings Of The IEEE* **92**(2), 271–293. Available from IEEEXplore.

B¨lcskei, H. & Zurich, E. (2006), 'Mimo-ofdm wireless systems: Basics, perspectives, and challenges', *IEEE Wireless Communications* pp. 31–37. Available from IEEEXplore.

Cvetković, Z. (1999), 'Modulating waveforms for ofdm', *1999 International IEEE Conference on Acoustics, Speech and Signal Processing* **5**, 2463–2466. Available from IEEEXplore.

Denny, M. (2007), *Blip, Ping, and Buzz: Making Sense of Sonar and Radar*, The

John Hopkins University Press, 2715 North Charles Street, Baltimore, Maryland 21218-4363, United States.

Dörpinghaus, M. & Speth, M. (1999), OFDM Receivers for Broadband-Transmission [Online], Phd, Lehrstuhl fssr Integrierte Systeme der Signalveraberitung, Walter-Schottky-Haus, Sommerfeldstrasse 24, D-52074 Aachen, Germany.
http://www.iss.rwth-aachen.de/Projekte/Theo/OFDM/www_ofdm.html
current November 2008.

Forney, G. & Costello, D. (2007), 'Channel Coding: The Road to Channel Capacity', *Proceedings of the IEEE* **95**(6), 1150–1177. Available from IEEEXplore.

Gault, S., Hachem, W. & Ciblat, P. (2007), 'Performance analysis of an ofdma transmission system in a multicell environment', *IEEE Transactions on Communications* **55**(4), 740–751. Available from IEEEXplore.

Gentile, K. (2003), *Fundamentals of Digital Quadrature Modulation*, Mobile Dev & Design, Penton Media, Inc.
http://rfdesign.com/images/archive/302Gentile40.pdf
current September 2009.

Gibson, J. D. (1999), *The Mobile Communications Handbook*, Vol. 2 of *The Electrical Engineering Handbook Series*, second edn, IEEE Press & CRC Press, Boca Raton, Florida.

Glisic, S. (2004), *Advanced Wireless Communications*, John Wiley & Sons, Ltd, The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England.
http://www3.interscience.wiley.com.ezproxy.usq.edu.au/cgi-bin/bookhome/110499375
[Electronic Resource] current May 2009.

Goldsmith, A., Jafar, S. A., Jindal, N. & Vishwanath, S. (2003), 'Capacity Limits of MIMO Channels', *IEEE Journal On Selected Areas In Communications* **21**(5), 684–702. Available from IEEEXplore.

Griffiths, I. (n.d.), Implementation of MIMO Wireless Communications [Online], Phd, The University of Newcastle, University Drive, Callaghan, NSW 2308 Australia.

http://www.eng.newcastle.edu.au/~c2104305/mimo.html current November 2008.

Hernando, J. M. & Pérez-Fontán, F. (1999), *Introduction to Mobile Communciations Engineering*, number 14 *in* 'Mobile Communications Series', Artech House, Boston, London.

Ibnkahla, M. (2005), *Signal Processing for Mobile Communications Handbook*, CRC Press LLC, 2000 N.W. Corporate Blvd., Boca Raton, Florida, 33431, United States.

IEE (1999), *Modulating Waveforms for OFDM*, Vol. 5 of *1999 IEEE International Conference on Acoustics, Speech, and Signal Processing*, Cvetkovic, Z., Phoenix, Az. Available from IEEEXplore.

Jiang, M. & Hanzo, L. (2007), 'Multiuser mimo-ofdm for next-generation wireless systems', *Proceedings of the IEEE* **95**(7), 1430–1469. Available from IEEEXplore.

Laroia, R., Li, J. & Uppala, S. (2004), 'Designing a mobile broadband wireless access network', *Signal Processing Magazine, IEEE* **21**(5), 20–28. Available from IEEEXplore.

Leis, J. (2002), *Digital Signal Processing — A MATLAB-Based Tutorial Approach*, Research Studies Press Ltd, 16 Coach House Cloisters, 10 Hitchin Street, Baldock, Hertfordshire, SG7 6AE, England.

Li, Y. & Sollenberger, N. (1999), Clustered OFDM with channel estimation for high rate wireless data, *in* '1999 IEEE International Workshop on Mobile Multimedia Communications, 1999. (MoMuC '99)', Dept. of Wireless Syst. Res., AT&T Labs., Red Bank, NJ, USA, IEEE, San Diego, CA, pp. 43–50. Available from IEEEXplore.

Macario, R. C. V. (1997), *Cellular Radio: Principles and Design*, second edn, Macmillan Press Ltd, Houndmills, Basingstroke, Hampshire RG21 6XS, United Kingdom.

Matteo, T. (2005), Windowed/Shaped OFDM and OFDM-OQAM: Alternative Multicarrier Modulations for Wireless Applications, Phd, Università Degli Studi Di Padova.

http://www.dei.unipd.it/~trivella/thesis.pdf
current May 2009.

Mennen, A. (2005), *It's Your Call: The Complete Guide to Mobile Phones — Money Saving Tips for Users*, Relianz Communications Pty Ltd, P.O. Box 82, Zillmere, Queensland 4034, Australia.

Rahmati, A. & Azmi, P. (2008), 'Iterative reconstruction of oversampled ofdm signals over deep fading channels', *4th European Conference on Circuits and Systems for Communications* pp. 289–294. Available from IEEEXplore.

Ramjee, P. & Shinsuke, H. (2003), *Multicarrier Techniques for 4G Mobile Communications*, Artech House universal personal communications series, Artech House, Incorporated.
http://site.ebrary.com.ezproxy.usq.edu.au/lib/unisouthernqld/docDetail.action?docID=10082035
current May 2009.

Steele, R. (1992), *Mobile Radio Communications*, Pentech Press Publishers, Graham Lodge, Graham Road, London, England.

Sullivan, N. P. (2007), *You Can Hear Me Now: How Microloans and Cell Phones are Connecting World's Poor to the Global Economy*, 1st edition edn, Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, United States.

Zheng, K., Huang, L., Li, G., Cao, H. & Dohler, M. (2008), 'Beyond 3g evolution', *IEEE Vehicular Technology Magazine* pp. 30–36. Available from IEEEXplore.

Appendix A

Project Specification

University of Southern Queensland

FACULTY OF ENGINEERING AND SURVEYING

ENG4111/2 Research Project
PROJECT SPECIFICATION

FOR: **Sarah Anne Hugo**

TOPIC: SIMULATION AND PERFORMANCE EVALUATION OF MIMO-OFDM FOR THE FOURTH GENERATION (4G) CELLULAR NETWORKS

SUPERVISOR: Dr Wei Xiang

SPONSORSHIP: Faculty of Engineering and Surveying

PROJECT AIM: To simulate and evaluate the principles of MIMO-OFDM techniques.

PROGRAMME: (Issue A, 24th March, 2009)

1) Research principles of OFDM and MIMO.

2) Determine how OFDM and MIMO work together as a MIMO-OFDM system.

3) Simulate that system using C/C++ and/or MATLAB to develop that simulation software.

4) Evaluate the perfomance of the simulated OFDM-MIMO system

5) Compare the simulated model to the theoretical model.

6) Further areas of improvement need to be identified for a HD grade to be granted.

As time permits:

7) Research associated areas (such as antennas, cellular networks, signal processing, etc.) to confirm the equations used and provide background information.

8) Perform a critical analysis of results to determine the suitability of the MIMO-OFDM technique to meet the demands of the 4G cellular network.

AGREED:

SAHugo (student) Wei Xi (supervisor)

Date: 24/03/ 2009 Date: 31 / 3/ 2009

Examiner/Co-examiner:
01/04/09

Appendix B

Additional Information

B.1 Types of Fading and Interference

B.1.1 ISI: Inter-Symbol Interference

When being transmitted and received, signals do not stay whole. They are broken up into parts, and these parts are known as symbols. It is these symbols that are being sent and transmitted back and forth, and that are then recombined into the final signal at the mobile. It is also these symbols that play a part in multipath interference and propagation (of Section 2.3.1).

ISI occurs when these symbols interfere, and cause interference with each other that has effects much like noise. It takes adaptive coding techniques, such as OFDM, to attempt to deal with such interference and try to reduce ISI.

B.1.2 Gaussian Fading

Gaussian fading occurs when the receiver/transmitter of the mobile — the mobile station (MS) — has a clear ***LOS***, or line of sight, to the transmitter aerial. In this case, there are no echoes — multipath — or effectively none. The technical term for such a channel is “Additive Gaussian White Noise” — low-level noise that is added onto the channel, of the Gaussian type, to simulate the noise of the system. Although it has been a matter of much discussion, it has been investigated heavily, nor used for the simulation of a 4G system.

The reasoning for this decision is simple. A mobile system is, at its heart, a terrestrial system. Yet gaussian fading has long been used to simulate satellite channels (see Section 2.4.1 onwards for the explanations of what a channel is). Their simulation of “clear LOS” applies most distinctly to situations were transmission and receiving occurs above distortions caused by the atmosphere — a location that only specialised cellular systems reach.

The decision was made early to simulate a “conventional” terrestrial cellular system, where the *transmission medium* is solely air. This removed the Gaussian model from

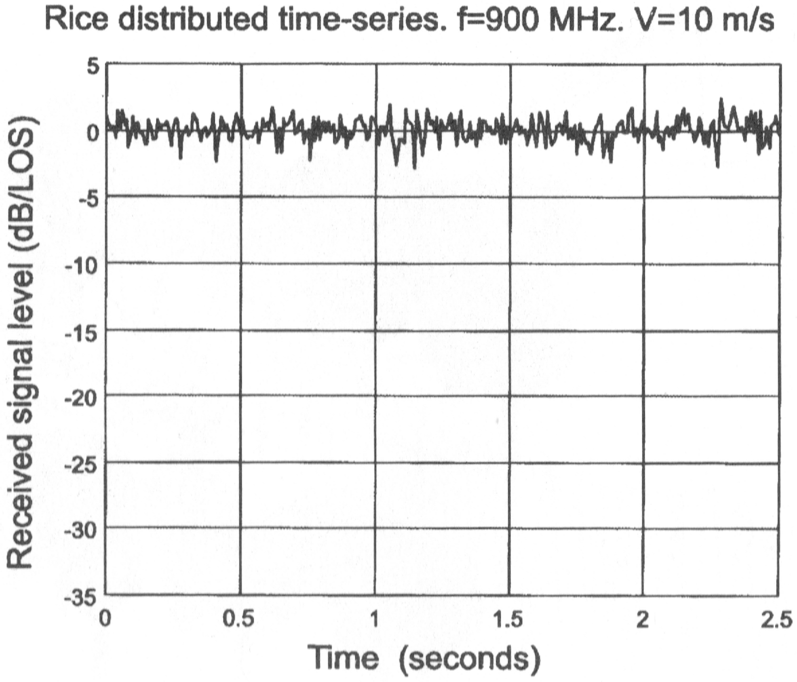

Figure B.1: Rice distributed time series. (Hernando & Pérez-Fontán 1999)

the equations, and kept the way open for Ricean and Rayleigh — a truly cellular model.

B.1.3 Ricean Fading

Ricean fading occurs when there is partial cancelation of the signal itself. That is, the signal arrives at the receiver by, say, two different paths, and at least one of the paths is stronger than the other. This stronger path is typically the LOS path.

In this case, the situation can be modeled by, appropriately, the Rice distribution. This is a continuous probability distribution, and describes (simulates) the expected value of the signal reaching the receiver due to the current parameters, It is given by:

$$p(r) = \frac{r}{\sigma^2} \exp\left(-\frac{r^2 + a^2}{2\sigma^2}\right) I_0\left(\frac{ra}{\sigma^2}\right) \qquad \text{for } r \geq 0 \tag{B.1}$$

where $p(r)$ is the probability density function, σ^2 is variance of the signal, r is the transmitted signal and a is that received.

This Ricean case is specified such a is not large, and a is not zero. As a gets closer to

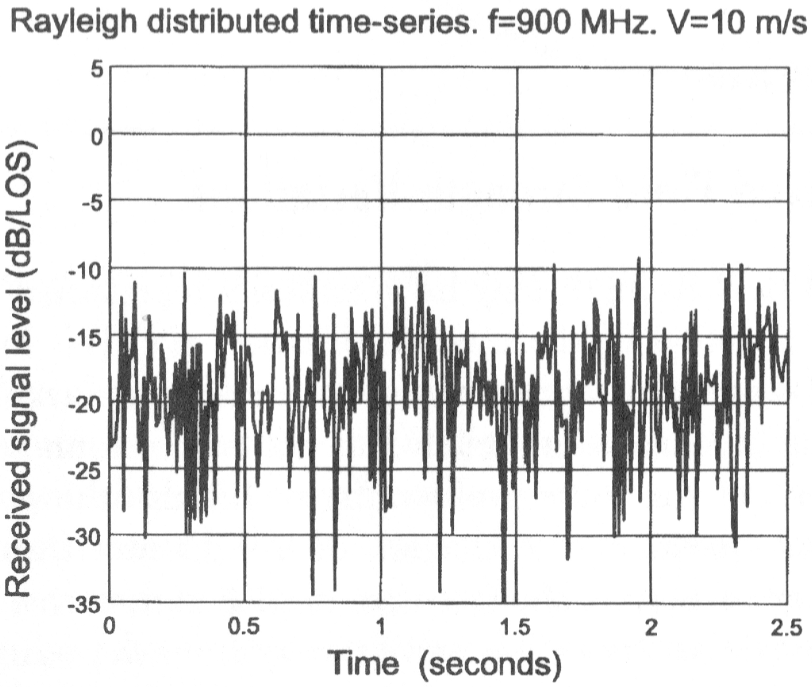

Figure B.2: Rayleigh distributed time series, relative to LOS. (Hernando & Pérez-Fontán 1999)

zero, the more Gaussian the fading is. As a gets large, the more Rayleigh fading would suit the situation.

B.1.4 Rayleigh Fading

Rayleigh fading occurs in heavily built-up urban environment — or a densely wooden rural area — where there is no LOS path to the aerial. Thus, there is no dominant signal (propagation) between the aerial and the receiver of the MS.[1]

The signal is attenuated, reflected, refracted, and diffracted by surrounding objects. This leads to a high amount of scatter, or variation.

[1] If there had been a more dominant signal from the antenna, Ricean fading would have been more applicable.

It is given by

$$p(r) = \frac{r}{\sigma^2} \exp\left(-\frac{r^2 + a^2}{2\sigma^2}\right) I_0\left(\frac{ra}{\sigma^2}\right) \qquad \text{for } r \geq 0 \tag{B.2}$$

where $p(r)$ is the probability density function, σ^2 is variance of the signal, r is the transmitted signal and a is that received.

Note, again, that it is the same formula as for the Ricean model. In this case, it models Rayleigh *if and only if* when a is large. In that case, the signal is not dominant, and there are large echoes.

Appendix C

Compilation and Output

As follows are some suggestions on various compilers to use with the program — or not to use, in some cases — if one wishes to compile the program themselves. Also offered in is proof of compilation, and Section C.3 gives some debug runs, to give a brief overview of how the program runs, although observers desiring more detail should look to Appendix D. There are also some thoughts in Section C.4 of how to convert the output of the data to graphical form without using a vendor-locked program.

C.1 Compilation in VC 2008 Express

This takes a little more work than plugging the code directly into a MinGW compiler/make setup.[1]

First, set the project with full optimisation. This may require turning off default runtime library (in the writer's version, that was the flag $/RTCx$, where x was su — the flag had to be changed to 'Default', or the equivalent). This also requires un-setting the flag for debug information being sent to a file/database. This is the flag $/Za$, $/Zi$ or various flavours thereof, hopefully found under 'General' — it has to be set to 'no' or 'inherit from parent or project'.

Attempt to turn off all options relating to "Unicode", and resolve further conflicts as needed.

As an option before compiling, add command line arguments: `4 4` — or, really any relatively small number below 1024. Then compile and run.

The end result: *time-wasting*. Be prepared to wait. Even a relatively simple run, with a 4 × 4 array *will* take some time. Running at full capabilities — a 1024 × 1024 array — will take at least 2 minutes, on average, just to initialise data. Other compilers can make the program do it a lot faster. Some rudimentary calculations from comparing the VC2008 compiler to others show it adds complexity of roughly $O(10n^n)$ — if it took 2 s in another compiler, it took 40 s in VC2008.

[1] Note that the following instructions are general, and somewhat tailored to the VC2008 release. They should be adapted to previous and later releases of the VC compiler.

In short, no matter how much the compiler is optimised, it will never be as fast as the G++ compilation. It has been tried with various compiler options, and this seemed to be a fact of life. This cannot be properly emphasised enough. Although the program *will* compile under VC++, the VC++ compiler is not a *true* C++ compiler as it did not properly implement the Standard C++ Library. Truer results are obtained by running the code through a compiler that holds more truly to the Standard C++ Library (also known as the C++ STL).

That is why the code was also compiled under MinGW in Windows and with the G++ compiler in Linux. When compilation is mentioned hereafter, consider it being done with the g++ compiler.

C.2 Compilation With MinGW and G++

The version that compiled under "DOS" via the Windows 'cmd' shell, was made through MinGW, and then again in Linux. Both attempts compiled. As follows is the MinGW output from a standard compilation run (with '>' standing in for the location in the hard-drive), using a custom-built make (see Section D.1 in Appendix D).

```
 > mingw32-make
g++    -c -o sim4G.o sim4G.cpp
g++    -c -o simulation.o simulation.cpp
g++    -c -o mimo.o mimo.cpp
g++    -c -o channel.o channel.cpp
g++    -c -o ofdm.o ofdm.cpp
g++    -c -o noise.o noise.cpp
g++    -c -o ber.o ber.cpp
g++    -c -o signal.o signal.cpp
g++    -c -o antenna.o antenna.cpp
g++    -c -o bitsstrm.o bitsstrm.cpp
g++    -c -o binary.o binary.cpp
g++    -c -o debug.o debug.cpp
g++    -c -o standard.o standard.cpp
g++    -c -o allvals.o allvals.cpp
```

```
g++ -o sim4G.exe sim4G.o simulation.o mimo.o channel.o ofdm.o noise.o ber.o sign
al.o antenna.o bitsstrm.o binary.o debug.o standard.o allvals.o

 >
```

C.2.1 Running The Program

As mentioned in Section C.1, the standard program came with a maximum array size of 1024 × 1024. This was to accommodate future technological advances, in both antennas (which is currently limited to 4 antennas in an array) and cpu cores (which at the time of writing, the maximum could vary between 8 to 16, although the Linux kernel has capability of managing 1024, hence the maximum value). This is also the number of times the program was (eventually going to be) able to fork.

Smaller array sizes therefore need to be specified at the command line, unless one desires to wait. For instance, as follows is the simulation under normal conditions, and then with a 10 × 10 construct...

```
>sim4G
Initialising data.
You have requested a 1024 by 1024 array.
This may take a while...

Running simulation...

Success! Exiting.


 >sim4G 10 10 1800
Initialising data.
You have requested a 10 by 10 array.
This may take a while...

Running simulation...
```

```
Success! Exiting.



>
```

A similar listing of the program running, this time with all debug statement turned on so as to show how it operates, follows. For the sake of simplicity, this output is a 1 × 1 array.

```
>sim4g 1 1 +
non-default run, parsing options
MIMO: NUMOFTX = 1 and NUMOFRX = 1

MIMO TRANSMITTER (mobile):
  1
  OFDM  channel aok, size = 1801
  Noise channel aok, size = 900
  BER   channel aok, size = 901
MIMO: Rx channel ok, bandwidth: 900
MIMO: Rx array created: 1

MIMO RECEIVER (base-station):
  1
  OFDM  channel aok, size = 1801
  Noise channel aok, size = 900
  BER   channel aok, size = 901
MIMO: Tx channel ok, bandwidth: 900
MIMO: Tx-array created: 1
Attempting forks.

printing to files.
everything deleted

>
```

The OFDM signal has a higher signal length, as it has an actual signal inside it —

one that is sampled at just over twice whatever the maximum bandwidth present in the system, which in this case, is 900. Noise, where length is not so critical, is exactly half, as is BER — which is more concerned with finding signal differences — and both signals can be easily adjusted in size regardless.

To show the flexibility of the system, the debug run is repeated this time with more antennas, and a higher bandwidth of 1900 (MHz).

```
>sim4g 10 10 1900 +
non-default run, parsing options
MIMO: NUMOFTX = 10 and NUMOFRX = 10

MIMO TRANSMITTER (mobile):
  1
  OFDM  channel aok, size = 3801
  Noise channel aok, size = 1900
  BER   channel aok, size = 1901
  2
  OFDM  channel aok, size = 3801
  Noise channel aok, size = 1900
  BER   channel aok, size = 1901
  3
  OFDM  channel aok, size = 3801
  Noise channel aok, size = 1900
  BER   channel aok, size = 1901
  4
  OFDM  channel aok, size = 3801
  Noise channel aok, size = 1900
  BER   channel aok, size = 1901
  5
  OFDM  channel aok, size = 3801
  Noise channel aok, size = 1900
  BER   channel aok, size = 1901
  6
  OFDM  channel aok, size = 3801
  Noise channel aok, size = 1900
  BER   channel aok, size = 1901
  7
  OFDM  channel aok, size = 3801
  Noise channel aok, size = 1900
  BER   channel aok, size = 1901
  8
  OFDM  channel aok, size = 3801
  Noise channel aok, size = 1900
  BER   channel aok, size = 1901
  9
  OFDM  channel aok, size = 3801
  Noise channel aok, size = 1900
  BER   channel aok, size = 1901
  10
  OFDM  channel aok, size = 3801
  Noise channel aok, size = 1900
  BER   channel aok, size = 1901
MIMO: Rx channel ok, bandwidth: 1900
MIMO: Rx array created: 10

MIMO RECEIVER (base-station):
  1
  OFDM  channel aok, size = 3801
  Noise channel aok, size = 1900
  BER   channel aok, size = 1901
  2
  OFDM  channel aok, size = 3801
  Noise channel aok, size = 1900
  BER   channel aok, size = 1901
  3
  OFDM  channel aok, size = 3801
  Noise channel aok, size = 1900
  BER   channel aok, size = 1901
  4
  OFDM  channel aok, size = 3801
  Noise channel aok, size = 1900
  BER   channel aok, size = 1901
  5
```

```
    OFDM  channel aok, size = 3801
    Noise channel aok, size = 1900
    BER   channel aok, size = 1901
    6
    OFDM  channel aok, size = 3801
    Noise channel aok, size = 1900
    BER   channel aok, size = 1901
    7
    OFDM  channel aok, size = 3801
    Noise channel aok, size = 1900
    BER   channel aok, size = 1901
    8
    OFDM  channel aok, size = 3801
    Noise channel aok, size = 1900
    BER   channel aok, size = 1901
    9
    OFDM  channel aok, size = 3801
    Noise channel aok, size = 1900
    BER   channel aok, size = 1901
    10
    OFDM  channel aok, size = 3801
    Noise channel aok, size = 1900
    BER   channel aok, size = 1901
MIMO: Tx channel ok, bandwidth: 1900
MIMO: Tx-array created: 10
Attempting forks.

printing to files.
everything deleted

>
```

This proves the program's ability to register more than one character at each option. Higher numbers could have been used, to further demonstrate the programs capability to implement a large-sized antenna array — up to 1024 × 1024 arrays were possible, if one so chose. It was not done here simply because of buffer over-runs (of output, not data capability) on the terminals being used.

One more interesting point, before handing over to actual output (instead of proof of compilation).

The standard the simulation held to, as it as so well documented, is the GSM bands, in terms of frequency location — that is, in full, 850 MHz, 900 MHz, 1800 MHz, and 1900 MHz. The standard one used, failing one supplied by the user, was 900 MHz — and if the user failed to hit one of the GSM bands, this was the band used — as this is the band most commonly used around the world at the time of writing.

C.3 Actual Output

As was mentioned in Chapter 6 and Chapter 7, the final code was subject to a shorted development, debugging, and implementation cycle — what some might call "rapid development". As such, there is, at the point of writing, technically no output being produced — beyond that obtained via checks through debugging statements and the like.

That being said, it can be guaranteed these three things:

- The "OFDM" signal *is* a sinusoidal signal, sampled at 1800 times a second.[2]
- The "Noise" signal *is* one of random noise.
- The "BER" signal *is* ready and waiting to find the difference between them.

Unfortunately, producing actual output such as might be converted into technical graphs had to be moved beyond the date of the dissertation submission. It might be emphasised, however, that what has been done, has been done in basically 2 months. For the those interested, a brief discussion of the actual output plans have been included in Section C.4.

[2] It is the modulation and demodulation where the IFFT and FFT take place. Hence the signal *does* need to have data, regardless of the modulation methods in use.

C.4 Converting Data to Graphical

The plans for converting the actual data of the simulation to a graph(s) had been all laid out. All that remained was the implementation — and the time to do so.

The plan was to use the svg standard, from the WW3 consortium — as has been already implemented in a number of graphical manipulators, the most notable of which is Inkscape[3]. The svg standard is used to create an image from a text file, which is perfect for a C++ file — where output is done into either text or binary. This would require adding a few extra modules to the program (if it was run in-situ), or creating a separating program (if it was run on the output file produced by the simulation). Either option would be possible.

A generic sketch of the proposed simulation's output file follows.

```
MOBILE
plot begin
    title   text
    xlabel  text
    ylabel  text
    point   [g,r,c]
plot end
group begin
    ber1.begin  ber2.begin  ber3.begin  ber4.begin
    ...         ...         ...         ...
    ber1.end    ber2.end    ber3.end    ber4.end
group begin
BASE
plot begin
    title   text
    xlabel  text
    ylabel  text
    point   [g,r,c]
plot end
```

[3] As a point of interest, the flowcharts, from Chapter 6, Section 6.2, were actually constructed using Inkscape.

```
group begin
    ber1.begin  ber2.begin  ber3.begin  ber4.begin
    ...         ...         ...         ...
    ber1.end    ber2.end    ber3.end    ber4.end
group begin
```

The [g,r,c] declaration is a hint to the order to look up the data in each 'group' or collection of data, i.e. group, then row, then column.

Here, ber1.begin and so on represent calls to the actual BER signals, to retrieve data, and so would be replaced in the file by actual numbers / values. The rest would appear as above, as actual text. This would be passed to the SVG program, wherever it occurs, to be converted into a SVG image of a graph in a relatively simple process — certainly much faster than feeding the values into MATLAB.

It also has the advantage that, if the option of a stand-alone program is used, the graph can be changed by various options on the command line. Simple!

Appendix D

Source Code

First, it should be clear that it was not easy to decide how to list the program. A program, especially one such as this, could not have a static model or approach to its development[1], and so its growth does not exactly fit most "computer science" models.

This gives rise to the following two questions. Should the listing be top-down, from main to the last, and thus lose a little in understanding what each function, even though this approach would highlight the execution flow of the program? Or should the listing be down-to-top, to reflect actual development, although this loses the benefit of execution flow? In the end, the former was the approach chosen, with data-types explained as they occur.

Secondly, an explanation of the programming choices.

Some standard files, used throughout the code, will be listed first. These were developed to aid the debugging process, and to prevent retyping (and relisting) of often used header files. As much as possible, actual code was removed from the header and placed in the implementation file. There are some case, in a few modules, were the implementation files are bare — reflecting the "still-in-development" nature of the program itself.

Something referred to quite frequently within the code, is the acronym "MOCA". It refers to the **M**IMO-**O**FDM-**C**hannel-**A**rray system that was implemented (as was explained in Chapter 6). It was, quite understandably, shortened to MOCA[2].

One final point is variable vs. function notation. All variables start with a lowercase letter, and functions start with an uppercase. Secondary words within names are capitalised, without underscores. This has commonly been referred to as "Hungarian" notation, in terms of the capitalization. Although in most cases there has been an effort to avoid inclusion of the name of the data type in the variable, and to instead strive for descriptive names, the former is probably the naming style more often used throughout the code.

Now for the files themselves.

[1] Especially in the debugging process, where modules seem to grow and change of their own accord.

[2] Being a computer program, though, it was most often used as "Moca" or "moca".

D.1 The makefile

Perhaps one of the best ways to get an overview of what's ahead is to look at what compiles, and how it compiles. At the risk of overwhelming some, here is the makefile. It is, however, a generic makefile. There is nothing "flashy" and stylish about this. It simply calls the compiler, hands it the files, and stands back. That is all there is to it.

There is, in total, 14 modules (15 if the main is counted as a separate module — which is a matter of semantics). Final dependencies were generated automatically through an option with the G++ compiler. This is the version included in the makefile below.

Listing D.1: The generic makefile.

```
# makefile
#
# Generic make file for Simulate4G
#
# Requires any make and the GCC compiler
#
# (c) by Sarah Hugo 2009
# Research Project 09-042
# Experimental Analysis for a 4G Mobile Network
# University of Southern Queensland

CC=g++
CFLAGS=-fpermissive -flax-vector-conversions -fimplicit-templates -x c++
LDFLAGS=
#-std=gnu++98
OBJECTS=sim4G.o simulation.o mimo.o channel.o ofdm.o noise.o ber.o \
  signal.o antenna.o bitsstrm.o binary.o debug.o standard.o allvals.o

sim4G.exe: $(OBJECTS)
    $(CC) -o sim4G.exe $(OBJECTS) $(LDFLAGS)

allvals.o: allvals.cpp allvals.h
antenna.o: antenna.cpp antenna.h debug.h standard.h allvals.h channel.h \
  noise.h signal.h bitsstrm.h ofdm.h
ber.o: ber.cpp standard.h debug.h signal.h bitsstrm.h ber.h
binary.o: binary.cpp binary.h debug.h standard.h
bitsstrm.o: bitsstrm.cpp standard.h bitsstrm.h debug.h binary.h
channel.o: channel.cpp channel.h debug.h standard.h allvals.h noise.h \
  signal.h bitsstrm.h ofdm.h
debug.o: debug.cpp debug.h standard.h
mimo.o: mimo.cpp mimo.h debug.h standard.h allvals.h antenna.h channel.h \
  noise.h signal.h bitsstrm.h ofdm.h
noise.o: noise.cpp standard.h debug.h signal.h bitsstrm.h noise.h
ofdm.o: ofdm.cpp standard.h debug.h signal.h bitsstrm.h ofdm.h
signal.o: signal.cpp signal.h debug.h standard.h bitsstrm.h
sim4G.o: sim4G.cpp standard.h simulation.h debug.h allvals.h mimo.h \
  antenna.h channel.h noise.h signal.h bitsstrm.h ofdm.h
simulation.o: simulation.cpp debug.h standard.h allvals.h simulation.h \
  mimo.h antenna.h channel.h noise.h signal.h bitsstrm.h ofdm.h
standard.o: standard.cpp standard.h


clean:
    rm *.o sim4G.exe


.PHONY: clean

#END OF MAKEFILE
```

D.2 The allvals module

This is where most (as many as possible) of the constants used throughout the program were defined. The reasons for this were pragmatic. It kept them in one place — it thus saved hunting through the files for one single value — and allowed for easy access and changing for debugging and, later, testing. (See Chapter 6 and Chapter 7 for details.

Listing D.2: Standard values: The 'allvals.h' header file.

```
/******************************************************************************
 * Sarah A Hugo
 * #09-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 *******************************************************************************
 * This code is freely available under the GNU General Public License.
 *******************************************************************************
 *  AllVals.h
 *     -- header file of values
 *        --> defines various important values for the simulation
 *              for later (easier) access.
 *        --> defines values for the Multiple-Antenna-Channel-Signals (MACS)
 *              system with easy access
 *******************************************************************************
 * Quick and easy definition of "global" simulation values, that are here for
 *   quick and easy access.
 * Similar to the Binary class in implementation, all I want is access to
 *   values, not so much 'objects', so its a *struct* with values, *not*  a
 *   class with values and functions.
 * That's why there's two 'struct's: one huge version for the simulation
 *   itself, and another "stripped down" version for the "MACS" system.
 *
 * SimVals and MacsVals can both be found here
 ******************************************************************************/

#ifndef _ALLSYSVALS_H_
#define _ALLSYSVALS_H_


namespace Sim4G
{
    /* SimVals:
     * public definitions of helper values for the simulation
     * for easy access elsewhere / at all places
     */
    struct SimVals
    {
        public:
            // pretend "constructor"
            explicit SimVals():
                // sim init
                maxProgOpts(4),
                isUsualRun(0), doIPrintHelp(1),
                areFilesGiven(2), isADebugRun(3),
                // antenna init
                txArrayIdx(0), rxArrayIdx(1),
                freqBandIdx(2), fileNameIdx(3),
                // antenna init vals
                GsmLoc(900),  maxArraySize(1024)
            {    }
            //----
            // Access to Values
            // Simulation.h/.cpp: about the program and/or user options:
            // maximum number of standard program relation options
            const size_t maxProgOpts;
            // for iterating through progOpts array (starting at 0)
            //progOpts
            //  = {isUsualRun, doIPrintHelp, areFilesGiven, isADebugRun}
            const size_t isUsualRun;
            const size_t doIPrintHelp;
            const size_t areFilesGiven;
            const size_t isADebugRun;
            // for iterating through userOpts vector (starting at 1)
            //userOpts
            //  = {iTxArraySize, iRxArraySize, freqBandIdx, fileNameIdx}
            const size_t txArrayIdx;
            const size_t rxArrayIdx;
            const size_t freqBandIdx;
            const size_t fileNameIdx;

            //----
            // Antenna.h/.cpp: antenna array constants
            // flexible value, for future technology (pipe dreams) *g*
            const size_t maxArraySize;
            // MHz location of GSM band: 900
            // (also used in simulation.h/.cpp)
            const size_t GsmLoc;
            // END of SimVals struct
    };




    /* because we only need a stripped down version of SimVals for the
     * MIMO-OFDM system with all the init vals, here's a smaller version
     * with less (yet "more", funnily enough) values
     * This value system also affects the channel and array system.
     * So its the Multiple-Antenna-Channel-Signals value system... or MACS. :)
     */
    struct MacsVals
    {
        public:
            // pretend "constructor
            explicit MacsVals():
                //init vals
                GsmLoc(900),  maxArraySize(1024), maxBandwidth(100),
                // Antenna array types
                mobileAr(1), baseAr(2), personalAr(3),
                // Channel signal indexes
                ofdmType(1), noiseType(2), berType(3)

            {    }

            //----
            // Antenna.h/.cpp: antenna array constants
            // flexible value, for future technology (pipe dreams) *g*
            const size_t maxArraySize;
            // MHz location of GSM band: 900
            // (also used in simulation.h/.cpp)
            const size_t GsmLoc;
            // max size of our antenna/channel constructs
            const size_t maxBandwidth;
            // arbitary values designating antenna types
            const size_t mobileAr;
            const size_t baseAr;
            const size_t personalAr;      // not yet implemented!!!

            //----
            // Channel.h/.cpp: channel constants
            // defines which type of signal dealing with
            const size_t ofdmType;
            const size_t noiseType;
            const size_t berType;
            // END of MacsVals struct
    };


}
// end of namespace in header

#endif
// end of _ALLSYSVALS_H_ header
```

Both `structs` in ”allvals” are simply initialised declared to have values in the header file. Therefore, there is no work for the implementation file. But as the header is not pre-compiled, it is included here for completeness and to ensure the compiler compiles and links the header file as it should. It also includes the call to the <cstdlib> header, for the size_t definition — as otherwise, the module would fail, as it is basically a stand-alone module.

Listing D.3: Standard values: the 'allvals.cpp' implementation file.

```
/****************************************************************************
 * Sarah A Hugo
 * #09-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 ****************************************************************************
 * This code is freely available under the GNU General Public License.
 ****************************************************************************
 *  AllVals.cpp
 *     -- implementation file of values
 *        --> defines various important values for the simulation
 *              for later (easier) access.
 *        --> defines values for the MIMO-OFDM-Channel-Array (MOCA)
 *              system with easy access
 ****************************************************************************
 * All values are in the header file, so need for anything here.
 ***************************************************************************/

#include <cstdlib>
#include "allvals.h"
```

D.3 The standard module

From prior programming experience, there have always been a number of standard headers that have been repeatedly called when crafting a module. This program was no different. To save time — and typing — there was this, one of the more important sections of the program. It simply provided a list of the routinely called header files used throughout the modules. As such, it was called in every module in some way, or was an implicit dependency — the only one that did not, in fact, was “allvals” of Section D.2.

Listing D.4: Standard header list: the ’standard.h’ list of headers.

```
/*******************************************************************************
 * Sarah A Hugo
 * #9-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 *******************************************************************************
 * This code is freely available under the GNU General Public License.
 *******************************************************************************
 *  Standard.h
 *     -- header file with typical include files
 *        --> things used frequently, but changed infrequently
 *******************************************************************************
 * Call this instead of retyping them out each time.
 * WARNING: These headers have been carefully searched through and checked
 *  to make sure these calls are accurate and needed. Please do NOT remove.
 ******************************************************************************/

#ifndef _STANDARD_H_BASE_
#define _STANDARD_H_BASE_

// generally implicitly included but some compilers don't, so just in case...
// uncomment the next line if you have the header...
//#include <libio.h>     // for the pesky NULL definition DO NOT REMOVE
#include <cmath>        // because we need to use sinf() for signals

// input/output
#include <stdio.h>      // foundation block of i/o
#include <iostream>     // c++ streams
#include <fstream>      // file streams for i/o
#include <sstream>      // string streams

// exceptions
#include <stdexcept>    // error types: because they happen
#include <cstddef>      // see above
#include <exception>    // ditto

// types
#include <string>       // c++ improvement on the char*
#include <deque>        // container
#include <vector>       // container
#include <iterator>     // what makes deques and vectors work

// namespaces...
using namespace std;

#endif
// end of _STANDARD_H_BASE_
```

And for completeness, as all headers (unless pre-compiled) need an implementation file to ensure the header is called...

Listing D.5: Standard header list: the 'standard.cpp' implementation file.

```
/******************************************************************
 * Sarah A Hugo
 * #9-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 ******************************************************************
 * This code is freely available under the GNU General Public
 *   License.
 ******************************************************************
 *   Standard.h
 *     -- header file with typical include files
 *        --> things used frequently, but changed infrequently
 ******************************************************************
 * Call this instead of retyping them out each time.
 *****************************************************************/

#include "standard.h"
```

D.4 The debug module

Also from prior programming experience, it was known that it was handy to have a way to print out debugging statements as one went through the code’s development cycle. As such, this was a standard header-implementation module for debugging often used in personal projects, that fitted neatly into the code — the main work was passing down the boolean values through function parameters from `main` right down to where it was needed.

This module will implement the debug statements (found scattered throughout the code). The debug statements, and this module, have been left in for the sake of completeness. Access to their output is achieved via running the programming as

```
> sim4g +
> ./sim4g.exe +
```

where `>` represents the console/terminal command line (be it Linux or Dos)[3]. Removing the `+` from the command line (it can be put there in any order among other command line inputs) will not print any debug statements to the ’console’ or ’terminal’.

When the program was ready for final release, the programmer simply has to remove the capability of the program to recognise the `+` switch (and the associated note in the help screen). The program will then assume default operation, with all debug statements turned off.

Listing D.6: Debuging statement streams: ’debug.h’ header file

```
/********************************************************************************
 * Sarah A Hugo
 * #9-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 ********************************************************************************
 * This code is freely available under the GNU General Public License.
 ********************************************************************************
 *  Debug.h
 *     -- debug class header file
 *         --> outputs debugging strings to a given strm
 *         --> default is cout, but other strms may be specified
 ********************************************************************************
 * Implements a debugging class. Turned 'on', it displays the debugging
 *   statements throughout the code to the ostream given to the class when it
 *   is implemented (default: std::clog). Turned 'off', these statements are
 *   not displayed, and the program functions as normal.
 * Use as per nomal cout/clog stream, but without specifing endline:
```

[3] Both syntaxes are provided for completeness.

```
 *   Debug(bool) << "" << val;
 ****************************************************************************
 * It is turned 'off' and 'on' by passing a command argument '+' to the program
 *   at runtime. The lack of it turns the statements 'off'. The presence of the
 *   '+' turns the statements 'on'.
 ****************************************************************************/

#ifndef _DEBUGGING_H_
#define _DEBUGGING_H_

// standard header -- for i/o, types, namespaces, and exceptions
#include "standard.h"


namespace Sim4G {

    // class for debugging
    class Debug {
        private:
            // disallow copying
            // private data member
            const bool doIPrint;            // print if true

        public:
            // public constructor and destructor
            Debug(bool turnItOnOff): doIPrint(turnItOnOff) { }
            // force going to new line at end of output
            ~Debug() {
                if (doIPrint) {
                    std::clog << std::endl;
                }
            }


            // handy overloading of an already overloaded operator
            // so we can output the debugging statements...
            // ...pass some string
            std::ostream& operator<< (const char someChars[]);
            // ...pass some type
            std::ostream& operator<< (void* itemToPass);

    };

}

#endif
// end of _DEBUGGING_H_ header
```

And the implementation file, where the main work on the output streams takes place...

Listing D.7: Debugging statement stream: 'debug.cpp' implementation file.

```
/****************************************************************************
 * Sarah A Hugo
 * #9-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 ****************************************************************************
 * This code is freely available under the GNU General Public
 *   License.
 ****************************************************************************
 *  Debug.h
 *     -- debug class header file
 *        --> outputs debugging strings to a given strm
 *        --> default is cout, but other strms may be specified
 ****************************************************************************
 * Relatively trivial details of the overloading of the '<<' operator...aka
 *   how it outputs to the console, and doesn't if it doesn't need to.
 ****************************************************************************/

#include "debug.h"

using Sim4G::Debug;


// handy overloading of an already overloaded operator
// so we can output the debugging statements
std::ostream& Debug::operator<< (const char someChars[]) {
    // if not requried to print (doIPrint is false)
    // to make sure we don't print if don't need to
    if (!doIPrint) {
        // basically do nothing, output nothing
        std::clog.setstate(std::clog.failbit);
        return std::clog;
    }
    // else add if something to print and output as we go
    else {
        for (size_t i=0; someChars[i]!='\0'; i++) {
            std::clog.put(someChars[i]);
        }
        return std::clog.flush();
    }
}

std::ostream& Debug::operator<< (void* itemToPass) {
    // if doIPrint is false, and not required to print
    // to make sure we don't print if don't need to
    if (!doIPrint) {
        // so basically do nothing, output nothing
        std::clog.setstate(std::clog.failbit);
        return std::clog;
    }
    // otherwise, add everything to strm and output as we go
    else {
        return std::clog << itemToPass;
    }
}
```

D.5 The Sim4G module

This is the main file. It might seem overwhelming, but it can be simply broken down down into two groups:

- A quick parse through the command line options to hand them over for simulation.
- Directing program flow:
  - Initialising data.
  - Running the simulation.
  - Handling errors from the above, and/or avoiding unwanted behavior.

The largest portion of work done is in the second group: direction program flow, in particular error catching and avoiding unwanted behavior. In error catching, this requires catching all the exceptions thrown throughout the subsequent modules, handling them, and returning appropriate information to both the user and the operating system. In terms of unwanted behavior, for instance, if the user requests help, it goes straight to the help screen function and then exits, without having to initialise data and run the simulation.

Being a C++ program, `main` is not the actual program. The actual program happens through the `Simulation` class — in `main`, its object is `sim` — which, in turn, is what actually handles initialising the data and running the program through `sim`'s functions. It is also `sim` that is responsible for throwing the errors that `main` has to handle.

Listing D.8: The main file: 'sim4g.cpp' implementation file.

```
/******************************************************************************
 * Sarah A Hugo
 * #09-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 ******************************************************************************
 * This code is freely available under the GNU General Public License.
 ******************************************************************************
 *  sim4G.cpp -- aka main.cpp
 *     -- source file for actual simulation
 *        --> starts everything running, handles it, and exits
 ******************************************************************************
 * Retrieves command line inputs, puts them into an options list for later
 *   parsing, then initialises and "runs" the simulation.
 * Also handles (catches) any thrown errors during the process of running said
 *   simulation, and returns the appropriate error value -- usually to the
 *   console, but can be any interfacing application -- via std::cout.
 * Other note: Simulation output goes into text files for later output into
 *   graphical form. Can be renamed from command line.
 ***************************************************************************/

// standard header for i/o, namespaces, exceptions, 'n types
#include "standard.h"
// my header for the simulation
#include "simulation.h"
// my header for debugging (optional display - default is no)
#include "debug.h"

using namespace Sim4G;


/* forward declaration of helper funcs:
 * pushes options into (optList and progOpts) from (argv and argc)
 */
void CreateOptList(int, char*[], std::vector<char*>&, bool[]);
/* Print helpful words on error, and tell the user to seek out help.
 * (and how to do so)
 */
void PrintFinalWords(bool myError);

/* Internal format of command-line input:
 * argv[0] => program path/name
 * argv[1] => where options start
 * argv[n] => nth option
 * argv[argc+1] = null0
 * int argc => total no. of commands/paths/names/options available
 */
int main(int argc, char* argv[])
{
    bool myError = false;

    // standard options
    // from Sim4G::PROGLIST
    // { isUsualRun, doPrintHelp, areFilesGiven, isADebugRun };
    bool myProgOpts[4] = { true, false, false, false };

    std::vector<char*> lineOptList; // list of options from user
    // pick up options from the user, and our standard opts
    CreateOptList(argc, argv, lineOptList, myProgOpts);

    // having checked options, set up simulation
    SimVals simStandards = SimVals();
    Simulation* sim = new Simulation();
    if (!sim) {
        std::cerr << "Failed␣to␣allocate␣memory␣for␣simulation."
                  << "␣Exiting." << std::endl;
        PrintFinalWords(myError = false);
        return -1;
    }
    // get user's/sim's options and commands from command line
    sim->MakeSimOpts(lineOptList, myProgOpts);

    // temp save of trigger values: debugging and default runs
    bool ynDebug = myProgOpts[simStandards.isADebugRun];
    bool isDefault = myProgOpts[simStandards.isADebugRun];

    // if need to print help, that's all we need to do
    if (myProgOpts[simStandards.doIPrintHelp]) {
        // print help, with (optional) extra embellishments
        sim->HelpScreen(ynDebug);
        return 0;
    }

    // run actual simulation and create data only when have to
    try {
        // because this may take a while, print something for user
        // trick: only print this on standard (non debug) run
        if (!ynDebug) {
            std::cout << "Initialising␣data." << std::endl;
        }
        sim->InitialiseData();
    } catch(logic_error) {
        // thrown by mimo on being unable to set up rx/tx array
        std::cerr << "MIMO␣error.␣Internal␣logic.␣Exiting." << std::endl;
        PrintFinalWords(myError = true);
        return -1;
    } catch(runtime_error) {
        //thrown on memory problems in Mimo, but it could be input (cmd line)
        std::cerr << "MIMO␣error.␣Memory␣or␣input." << std::endl;
        PrintFinalWords(myError = false);
        return -2;
    }

    // all data initialised, safe to run the simulation

    // again, print a helpful message for the user
    // that appears on a standard run
    if (!ynDebug) {
        std::cout << std::endl;
        std::cout << "Running␣simulation..." << std::endl;
    }
    sim->Run(isDefault, ynDebug);

    // space here to "feed" a presentation func the sim object
    //      aka Sim4GPres(sim);
    //      aka Sim4GTextOut(sim);
    //      aka Sim4GSvgOut(sim);
    // *before* the object is destroyed...if you want to.

    // everything run ok, so exit
    if (!ynDebug) {
        std::cout << std::endl;
        std::cout << "Success!␣Exiting." << std::endl;
        std::cout << std::endl;
    }
    delete sim;
    return 0;
}

/* printer: what to print when something wrong happens
 * depending on if its the user’s fault or my fault
 * AKA optimally, you should never these words
 */
void PrintFinalWords(bool myError)
{
    std::cout << std::endl;
    // on error, its our fault
    if(myError) {
        std::cout << "Oops.␣I␣seem␣to␣have␣made␣a␣mistake␣somewhere!"
                  << std::endl;
    } else {
        // otherwise, its their fault
        std::cout << "Oh␣dear,␣you’ve␣made␣a␣mistake␣somewhere!"
                  << std::endl
                  << "Perhaps␣you␣gave␣me␣the␣wrong␣data." << std::endl
                  << "Or␣your␣memory␣needs␣some␣attention." << std::endl;
    }
    std::cout << "Use␣the␣help␣screen␣that␣comes␣with␣this␣program␣for␣advice."
              << std::endl
              << "Try␣using␣‘--help’␣or␣‘--?’␣as␣options,␣next␣time,␣okay?"
              << std::endl
              << "Thanks." << std::endl;

}


/* helper function...
 * add options to vector of Options List
 */
void CreateOptList(int argc, char* argv[],
                   std::vector<char*>& optList, bool progOpts[])
{
    // standard options
    SimVals simStandards = SimVals();
    size_t maxProgs = simStandards.maxProgOpts;
    // add onto the optList the program-related options
    if (argc > 1) {
        // get here if (at least) one user-provided option available
        // set to: non-standard-run, no-print-help,
        //          no-files-given, old-debug-status
        bool oldDebugState = progOpts[simStandards.isADebugRun];
        for (size_t i=0; i<maxProgs; i++) {
            progOpts[i] = false;
        }
        progOpts[simStandards.isADebugRun] = oldDebugState;
        // now we iterate through available options
        int i = 1;
        do {
            // option available, automatic entry into opt-list
            optList.insert(optList.end(), argv[i]);
            // check if provided filenames for output
            if (argv[i][0] == ’-’ && argv[i][1] != ’-’) {
                // adjust progOpts and our optionslist accordingly
                progOpts[simStandards.areFilesGiven] = true;
            } else if (argv[i][0] == ’-’ && argv[i][1] == ’-’) {
                // request for help, so trigger and rmeove
                progOpts[simStandards.doIPrintHelp] = true;
                optList.pop_back();
            } else if (argv[i][0] == ’+’) {
                // debug statements request, so trigger and remove
                progOpts[simStandards.isADebugRun] = true;
                optList.pop_back();
            }
            // option handled, move onto next option/entry
            i++;
        } while (argv[i] != NULL);
    }
}
```

D.6 The Simulation module

The Simulation class. It takes the list of options from the command line and parses them into something useful (MakeSimOpts, using optList from main and simStandards, of type SimVals from Section D.2), thus making sure that what was received from the command line as options is within standard, acceptable ranges (i.e. GSM frequency range, size of antenna array(s), standard output file, et cetera).

Simulation also has a SimOut function, which is where the actual output to the file takes place.

Listing D.9: The controlling 'Simulation' class: 'simulation.h' header file.

```
/*******************************************************************************
 * Sarah A Hugo
 * #09-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 *******************************************************************************
 * This code is freely available under the GNU General Public License.
 *******************************************************************************
 *  Simulation.h
 *     -- header file for simulation
 *        --> defines / controls everything
 *******************************************************************************
 * Overall running and initialisation of the simulation. Also takes care of
 *  overall input/output, and error/exception catching.
 ******************************************************************************/

#ifndef _SIMULATION_H_
#define _SIMULATION_H_


// for debuging
#include "debug.h"
// standard header -- for i/o, types, namespaces, and exceptions
#include "standard.h"
// for standard simulation values
#include "allvals.h"
// for the multiple antenna array definition
#include "mimo.h"


namespace Sim4G
{
    class Simulation
    {
    private:
        // standard simulation values
        SimVals simStandards;
        // the list of program-related options
        // via simStandards:{isUsualRun,doIPrintHelp,areFilesGiven,isADebugRun}
        bool progOpts[4];
        // a list of opts from command line to parse (vary with each run)
        // via simStandards:{rxArraySize, txArraySize, freqBand, fileNames}
        vector<char*> userOpts;
        // the actual options passed by the user
        std::string binFiles;        // string to hold the users filenames
        size_t numOfTx;              // number of transmitter antennas
        size_t numOfRx;              // number of receiver antennas
        size_t freq;                 // frequency band to operate at
        // parses simulation opts to get them into a useable form
        // throws exceptions if encounters major errors
        void ParseCommLine(vector<char*>&, bool*);
        // gets the user opts (from command line) into more useable forms
        void RetrieveSimOpts();

        // data members (that hold all the functions we need)
        Mimo* allAntennas;           // antennas to transmit/receive
        // Functions for basic running of simulation...
        void TransmitReceive();     // send signals between antennas
        void FindErrors();          // results of sending/receiving (BER)
        void SimOut(bool);          // output of simulation data

    public:
        // Functions to create and destory the basic simulation
        Simulation()
        {
            // get the standard values for later use
            SimVals simStandards = SimVals();
        }
        ~Simulation();
        // Function to get simulation options in order
        // using data from command line (from main->sim4g.exe)
        void MakeSimOpts(vector<char*>&, bool*);
        // Once parsing completed, initialise data
        void InitialiseData();
        // Function to display help if needed
        void HelpScreen(bool);
        // Function to actually *run* the simulation
        // (with forking, and bools for default run and doing debugging)
        void Run(bool, bool);
    };

    // end of namespace 4GSim (for now)
}
// end of _SIMULATION_H_
#endif
```

And the implementation file. Note how much of the actual function definitions actually take place in the implementation file. This is the true modular style, and is what is done as much as possible.

The SimOut function was designed to set up the file for output, pass the file down the simulation to where the output would actually take place, receive the results, then close the file. It has not, as this stage, been implemented, nor has its associated functions been implemented, and so its contents are commented out.

Listing D.10: The controlling 'Simulation' class: 'simulation.cpp' implementation file.

```
/******************************************************************************
 * Sarah A Hugo
 * #09-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 ******************************************************************************
 * This code is freely available under the GNU General Public License.
 ******************************************************************************
 *  Simulation.h
 *     -- header file for simulation
 *        --> defines / controls everything
 ******************************************************************************
 * Overall running and initialisation of the simulation. Also takes care of
 *  overall input/output, and error/exception catching.
 *****************************************************************************/

// standard headers
#include "debug.h"
#include "standard.h"
// my headers
#include "allvals.h"
#include "simulation.h"
#include "channel.h"
#include "mimo.h"

using Sim4G::Debug;
using Sim4G::SimVals;
using Sim4G::Simulation;
using Sim4G::Channel;
using Sim4G::Mimo;


/* parsing option list to make them usable for the simulation:
 * equates given optlist (from sim4g [main]) to classes [this], and
 * given progOpts (from main) to classes (this).
 */
void Simulation::ParseCommLine(std::vector<char*>& optList, bool* givenPOpts)
{
    this->userOpts.swap(optList);
    // get out the first four options
    size_t maxSize = this->simStandards.maxProgOpts;
    for (size_t i=0; i<maxSize; i++) {
        this->progOpts[i] = givenPOpts[i];
    }
}

/* Gets the command line opts from 'userOpts vector into the classes own
 * variables.
 * Takes no arguments, returns none.
 */
void Simulation::RetrieveSimOpts()
{
    // to accept user options...
    size_t numUserOpts = this->userOpts.size();

    // parse through the inputs, handling more and more inputs in turn
    // using cascading if statements
    // ugly fix but it *works*... trust me on this one

    // at least 4 input extra input:
    // filenames provided, by user
    if (numUserOpts > this->simStandards.fileNameIdx) {
        this->binFiles = this->userOpts.at(this->simStandards.fileNameIdx);
    } else {
        // but if filename not provided...get default value
        this->binFiles = "output.tdt"; // file: Text Deliminated by Tabs
    }

    // at least 3 extra input:
    // frequency band to use provided, by user
    if (numUserOpts > this->simStandards.freqBandIdx) {
        // allocate to user option and make sure its a GSM band
        this->freq = atoi(this->userOpts.at(this->simStandards.freqBandIdx));
        if (!(this->freq==850 || this->freq==900 || this->freq==1800
            || this->freq==1900)) {
            // if not within GSM range, make standard GSM band
            this->freq = this->simStandards.GsmLoc;
        }
    } else {
        // but if frequency not provided... get default value
        this->freq = this->simStandards.GsmLoc;
    }

    // at least 2 extra input:
    // at least both antenna arrays, by user
    if (numUserOpts > this->simStandards.rxArrayIdx) {
        // convert char to number
        this->numOfRx = atoi(this->userOpts.at(this->simStandards.rxArrayIdx));
        // make sure value is within limits of technology
        if (this->numOfRx > this->simStandards.maxArraySize) {
            this->numOfRx = this->simStandards.maxArraySize;
        }
    } else {
        // but if receiver array not providedget default value
        this->numOfRx = this->simStandards.maxArraySize;
    }

    // at least 1 extra input:
    // number of antennas (array) at Transmitter (Tx) provided (max 4)
    if (numUserOpts > this->simStandards.txArrayIdx) {
        // convert char to number
        this->numOfTx = atoi(this->userOpts.at(this->simStandards.txArrayIdx));
        // make sure value is within limits of technology
        if (this->numOfTx > this->simStandards.maxArraySize) {
            this->numOfTx = this->simStandards.maxArraySize;
        }
    } else {
        // but if num of transmitter array not provided... get default value
        this->numOfTx = this->simStandards.maxArraySize;
    }
    // if 0 inputs, don't need to do anything
    // default values already handled
}

// Controller function for parsing simulation data from command line
void Simulation::MakeSimOpts(vector<char*>& optList, bool* givenProgOpts)
{
    // get options into useable form for simulation
    ParseCommLine(optList, givenProgOpts);
    // safe to proceed with simulation, able to run as user desired
    // so get what we need to run
    RetrieveSimOpts();
}

// actually initialise the data
void Simulation::InitialiseData()
{
    // create antennas with capability of sending signals between them
    bool ynDebug = this->progOpts[this->simStandards.isADebugRun];
    if (this->progOpts[this->simStandards.isUsualRun]) {
        try {
            // is a default run
            Debug(ynDebug) << "usual␣run,␣default␣options";
            // setup as per normal: max mobiles, max base-stations :)
            Mimo* allAntennas = new Mimo(this->numOfRx, this->numOfTx,
                                         this->freq, ynDebug);
        } catch(logic_error) {
            // thrown by mimo on being unable to setup rx/tx array
            // in big trouble, unable to continue
            throw(logic_error("Setup.␣Unable␣to␣continue.␣Exiting"));
        } catch(runtime_error) {
            // not enough memory, still can't continue
            throw(runtime_error("Memory.␣Unable␣to␣continue.␣Exiting."));
        }
    } else {
        Debug(ynDebug) << "non-default␣run,␣parsing␣options";
        // provide antennas with user values
        try {
            // setup with user defined vals, my-defined debug statement trigger
            Mimo* allAntennas = new Mimo(this->numOfTx, this->numOfRx,
                                         this->freq, ynDebug);
        } catch(logic_error) {
            // thrown by mimo on unable to setup receiver/transmitter array
            // in big trouble, unable to continue
            throw(logic_error("Setup.␣Unable␣to␣continue.␣Exiting"));
        } catch(runtime_error) {
            // not enough memory, still can't continue
            throw(runtime_error("Memory.␣Unable␣to␣continue.␣Exiting."));
        }
    }
}

// save data and destroy the simulation's objects
Simulation::~Simulation()
{
    bool yn = this->progOpts[this->simStandards.isADebugRun];
    Debug(yn) << "everything␣deleted";
    // destroy the antennas and channel information
    // because everything is in deques, the deques handle the memory
    // and we don't have to do anything
}

// Disply help to standard out (cout) about application if requested
void Simulation::HelpScreen(bool doIPrint)
{
    // :) Print extra embellishments if the 'debug' mode is turned on
    bool yn = doIPrint;
    Debug(yn) << "****************************************";
    std::cout << "this␣is␣the␣help␣screen." << std::endl;
    Debug(yn) << "*******************************************";
    Debug(yn) << "␣";
    std::cout << "this␣is␣the␣list␣of␣inputs" << std::endl;
    std::cout << "" << std::endl;
}

// Info of how it runs...
void Simulation::Run(bool ynDefRun, bool turnItOn)
{
    Debug(turnItOn) << "Attempting␣forks.";
    Debug(turnItOn) << "␣";
    //fill antenna signals and fork
    // SendSig(); on each antennas
    // wait for forks to finish
    // check results
    //mimo.CheckBER();
    // stop forking
    // all done, so output and return
    SimOut(turnItOn);
    return;
}



// Function to provide output from the simulation's results...
void Simulation::SimOut(bool isADebugRun)
{

    Debug(isADebugRun) << "printing␣to␣files.";
/*    MacsVals macsStandards = MacsVals();

    // open text file

    size_t fadetype;
    if (fadetype == macsStandards.RiceFade) {
        // input results into file
    } else if (fadetype == macsStandards.RayleighFade) {
        // input results into file
    } else {
        // do nothing
        // or provide error (?)
    }

    // close text file

    // error check throughout!!
*/
}



// end of implementation file
```

D.7 The Mimo module

This is the crux of the MACS system mentioned previously in the dissertation.

The Channel and Antenna types, that are inserted into the deques (pronounced “deck”s), are as the name suggests, the containers for the channel and antenna implementations respectively. Channel, in particular, holds all signals for each Antenna. These deques are the most important containers in the program, and they cannot fail — hence the fact that Mimo throws exceptions if it detects errors in the setup process. Notes for how accessing each channel relates to each antenna are in the file’s top comment block.

Listing D.11: The overall ’Mimo’ class: ’mimo.h’ header file.

```
/******************************************************************************
 * Sarah A Hugo
 * #09-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 ******************************************************************************
 * This code is freely available under the GNU General Public License.
 ******************************************************************************
 *  Mimo.h
 *     -- header file for multiple input multiple output
 *        --> class definition
 *        --> the Mimo-Ofdm-Channel-Array system in action
 ******************************************************************************
 * Sets up and initialises the entire system, from the Channels (with its
 *   signal system) to the Antenna array system.
 * This is what is called in Simulation, and this is what calls everything
 *   else. In other words, if this breaks...RUN. Quickly.
 ******************************************************************************
 * HOW TO OPERATE (aka the implicit link):
 * When accessing an antenna, the same deque index ( *.at() function) will give
 *   you the antenna's channel, and vice versa. It's very simple.
 * To make it easier for yourself, you *could* make a function that, given an
 *   index, could return either depending on your return type --- but genuine
 *   C++ compilers *will* complain about this. Remembering the index trick is,
 *   in the end, easier.
 * **So do not forget**
 ****************************************************************************/

#ifndef _MIMO_H_
#define _MIMO_H_

// for debuging
#include "debug.h"
// standard header -- i/o, types, exceptions, namespaces
#include "standard.h"
// standard antenna related values
#include "allvals.h"
// the antenna definition
#include "antenna.h"
// the channel definition
#include "channel.h"


namespace Sim4G
{

    class Mimo
    {
    private:
        MacsVals macsStandards;
        // number of antennas in array
        const size_t NUM_OF_RX;
        const size_t NUM_OF_TX;
        // arrays of antennas at receiver/transmitter
        deque<Antenna> baseArray;
        deque<Antenna> mobArray;
        // channel used to send signals
        deque<Channel> airForMobiles;
        deque<Channel> airForBaseStats;

        // private helper functions
        Channel SetUpChannel(size_t, size_t, size_t, bool);
        void SetUpReceiverArray(size_t, size_t, bool);
        void SetUpTransmitterArray(size_t, size_t, bool);

    public:
        //set up the class
        explicit Mimo(size_t numTx, size_t numRx, size_t freq, //...
            bool doIDebug): NUM_OF_TX(numTx), NUM_OF_RX(numRx)

        {
            // construct list of vals.
            MacsVals macsStandards = MacsVals();

            // if on a standard run, print something helpful for the user
            // b/c this may take a while
            if(!doIDebug) {
                std::cout << "You␣have␣requested␣a␣" << this->NUM_OF_TX
                          << "␣by␣" << this->NUM_OF_RX << "␣array."
                          << std::endl;
                std::cout << "This␣may␣take␣a␣while..." << std::endl;
                std::cout << std::endl;
            }

            // set up receiver channel and antenna
            Debug(doIDebug) << "MIMO:␣NUMOFTX␣=␣" << this->NUM_OF_TX
                << "␣and␣NUMOFRX␣=␣" << this->NUM_OF_RX;

            // catch thrown errors and throw back out to simulation
            try {
                SetUpReceiverArray(NUM_OF_RX, freq, doIDebug);
            } catch(logic_error) {
                throw(logic_error("Unable␣to␣set␣up␣receiver␣array."));
            } catch(runtime_error) {
                throw(runtime_error("Memory␣allocation␣problems."));
            }

            // catch thrown errors and throw back out to simulation
            try {
                SetUpTransmitterArray(NUM_OF_TX, freq, doIDebug);
            } catch(logic_error) {
                throw(logic_error("Unable␣to␣set␣up␣receiver␣array."));
            } catch(runtime_error) {
                throw(runtime_error("Memory␣allocation␣problems"));
            }
        }
        // Destructor --> safe to use, no static members employed
        ~Mimo()
        {
            // by destroying antennas, we also destroy the channels
            // NOTE TO OTHERS: deques handle own memory, no need for ’delete’
        }
        // check it exist
        bool Exists()
        {
            return !(baseArray.empty() || mobArray.empty());
        }
        // output results / data in mimo system
        // input: filename to output channel info into
        void MimoSysOutput(std::string, bool doIDebug);

        // access tx’s and rx’s arrays (USE WITH CAUTION!!)
        deque<Antenna>* GetbaseArray()
        {
            return &baseArray;
        }
        deque<Antenna>* GetmobArray()
        {
            return &mobArray;
        }

        // number of antennas in the rx and tx arrays
        size_t NumOfbaseArray()
        {
            return NUM_OF_TX;
        }
        size_t NumOfmobArray()
        {
            return NUM_OF_RX;
        }
    };
    // end of namespace for now
}
```

```
// end of _MIMO_H_
#endif
```

And the implementation file. This is where the work is done, to set up each `Channel` and `Antenna`, according to the requested numbers of receivers and transmitters. Each receiver and transmitter has a separate setup function — it could have been more streamlined by making a 'generic' function, but this was more a micro-optimisation than a macro-optimisation. In this case, the decision was made to go with what worked and leave final 'tweaks' for afterwards.

Listing D.12: The overall 'Mimo' class: 'mimo.cpp' implementation file.

```
/*****************************************************************************
 * Sarah A Hugo
 * #09-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 *****************************************************************************
 * This code is freely available under the GNU General Public License.
 *****************************************************************************
 *  Mimo.cpp
 *     -- source file for setting up multiple input, multiple output
 *        --> all the equations involved, multi rx, multi tx
 *        --> aka the Multiple-Antenna-Channel-Signals (MACS) system at work
 *****************************************************************************
 * Sets up and initialises the entire system, from the Channels (with its
 *   signal system) to the Antenna array system.
 * This is what is called in Simulation, and this is what calls everything
 *   else. In other words, if this breaks...Run.
 ****************************************************************************/

// standard files
#include "debug.h"
#include "standard.h"
// my implementation files
#include "mimo.h"
#include "channel.h"
#include "antenna.h"
#include "allvals.h"

// general classes
using Sim4G::Debug;
using Sim4G::Mimo;
using Sim4G::Antenna;
using Sim4G::Channel;


Channel Mimo::SetUpChannel(size_t antennaArray, size_t numInArray,
                           size_t freq, bool doIDebug)
{
    // set up channel
    Debug(doIDebug) << "␣";
    if (antennaArray == this->macsStandards.mobileAr) {
        Debug(doIDebug) << "MIMO␣TRANSMITTER␣(mobile):";
    }
    else  {
        Debug(doIDebug) << "MIMO␣RECEIVER␣(base-station):";
    }
    return (Channel(numInArray, doIDebug, freq));
}


// how to setup an antenna array for receiver (mobile) in 3 easy steps
void Mimo::SetUpReceiverArray(size_t numInArray, size_t freq, bool doIDebug)
{
    // setup the channel
    Channel tmp = SetUpChannel(this->macsStandards.mobileAr, numInArray,
                               freq, doIDebug);
    // failsafe
    if (tmp.Healthy()) {
        this->airForMobiles.insert(this->airForMobiles.end(), tmp);
    } else {
        Debug(doIDebug) << "Failure␣to␣setup␣Rx␣channel.␣RUN!";
        throw std::logic_error("Rx␣channel␣failed");
    }
    Debug(doIDebug) << "MIMO:␣Rx␣channel␣ok,␣bandwidth:␣" << freq;

    // create antenna entry in rx deck give it the created channel
    // insert (at end of antArray) the (last channel-sys created)
    for (size_t i=0; i<numInArray; i++) {
        this->mobArray.push_back(Antenna(i, this->macsStandards.mobileAr,
                                         freq, this->NUM_OF_RX, doIDebug));
    }
    // should not happen, but just in case (memory problems)
    if(mobArray.empty()) {
        throw runtime_error("Array␣empty.");
    }
    Debug(doIDebug) << "MIMO:␣Rx␣array␣created:␣" << mobArray.size();
    return;
}

// how to setup an antenna array for transmitter (basestation) in 3 easy steps
void Mimo::SetUpTransmitterArray(size_t numInArray, size_t freq, bool doIDebug)
{
    // setup the channel
    Channel tmp = SetUpChannel(this->macsStandards.baseAr, numInArray,
                               freq, doIDebug);

    // failsafe
    if (tmp.Healthy()) {
        this->airForBaseStats.insert(this->airForBaseStats.end(), tmp);
    } else {
        Debug(doIDebug) << "Failure␣to␣setup␣Tx␣channel.␣RUN!";
        throw std::logic_error("Rx␣channel␣failed");
    }
    Debug(doIDebug) << "MIMO:␣Tx␣channel␣ok,␣bandwidth:␣" << freq;

    // create antenna entry in rx deck give it the created channel
    // insert (at end of antArray) the (last channel-sys created)
    for (size_t i=0; i<numInArray; i++) {
        this->baseArray.push_back(Antenna(i, this->macsStandards.baseAr,
                                          freq, this->NUM_OF_TX, doIDebug));
    }
    // should not happen, but just in case (memory problems)
    if(mobArray.empty()) {
        throw runtime_error("Array␣empty.");
    }
    Debug(doIDebug) << "MIMO:␣Tx-array␣created:␣" << baseArray.size();
    return;
}
```

D.8 The Antenna module

Notice that although the Antenna has a close tie with Channels, it itself has no reference to an actual Channel object (capitalisation, at this point, refers to the simulation implementation). Any reference to one caused errors, and so was removed. It does, however, have a counter, named channelNo, that can be accessed by others (namely Mimo, Section D.7) to relate the two.

Listing D.13: The overall 'Antenna' class: 'antenna.h' header file.

```
/*******************************************************************************
 * Sarah A Hugo
 * #09-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 ******************************************************************************
 * This code is freely available under the GNU General Public License.
 ******************************************************************************
 *  Antenna.h
 *     -- header file for the antenna characteristics
 *        --> overall look at the antennas and how they work in
 *            a 4G system.
 ******************************************************************************
 * Makes an antenna "struct", for placing *within* a deque, as happens in Mimo.
 * FOR FUTURE REFERENCE: This is NOT a class. It only has a "pretend" creator
 * to make things easier.
 * Do NOT explicitly delete. Let the deques take care of it, and be happy.
 * Could not be easier.
 ******************************************************************************/


#ifndef _ANTENNA_H_
#define _ANTENNA_H_

// for debuging
#include "debug.h"
// standard header -- i/o, types, exceptions, namespaces
#include "standard.h"
// for standard antenna/signal values
#include "allvals.h"


namespace Sim4G
{
    // the Antenna class of characteristics
    struct Antenna
    {
    private:
        // restrict access to generic creation
        MacsVals macsStandards;
        Antenna(void) {}
        // private data members of class
        size_t channelNo;       // relates Antenna to Channel
        size_t frequency;       // frequency of operation (MHz)
        size_t antType;         // type of station designated by antenna
        size_t arraySize;       // number of antennas around it in array
    public:
        Antenna(size_t count, size_t typeOfStation, size_t HzBand,
                size_t arraynum, bool doIDebug)
        {
            MacsVals macsStandards = MacsVals();
            //decide how to set up each antenna...
            channelNo = count;
            arraySize = arraynum;
            antType = typeOfStation;
            frequency = HzBand;
            // set up antenna types
            if (antType == this->macsStandards.mobileAr) {
                // set up mobile
                // particular parameters of the mobile
            } else if (antType == this->macsStandards.baseAr) {
                // set up base station
                // particular parameters of the basestation
            } else {
                // set up "personal antenna"
                // no accomodation as yet for "personal antennas"
                // ie antennas fitted to vehicles
            }
        }/*
        // copy constructor -- VITAL b/c its a *struct* not a class
        Antenna(const Antenna&)
        {}*/

        // relate Antenna to Channel
        size_t GetChannelNo();
        // function to output antenna information into strm
        void AntennaOutput(std::ofstream& outstrm);
    };
    // end of namespace Sim4G (for now)
}

//end of _ANTENNA_H_
#endif
```

And the implementation file.

Listing D.14: The overall 'Antenna' class: 'antenna.cpp' implementation file.

```
/******************************************************************************
 * Sarah A Hugo
 * #09-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 ******************************************************************************
 * This code is freely available under the GNU General Public License.
 ******************************************************************************
 *  Antenna.cpp
 *     -- implementation file for the antenna
 *        --> details of implementation at the antennas and how they work in
 *            a 4G system.
 ******************************************************************************
 * As Antenna has been made a struct (for insertion in 'deque's in Mimo),
 *   there is very little implementation (at this stage) in here.
 * Further development might see this page more full, and so it is included
 *   for completeness --- and the compiler.
 *****************************************************************************/

#include "antenna.h"

using Sim4G::Antenna;
using Sim4G::Debug;


/* GetChannelNo()
 * The relationship between Antenna and Channel, for Mimo.
 * Takes no arguments, returns a size_t counter.
 */
size_t Antenna::GetChannelNo()
{
    return channelNo;
}
```

D.9 The `Channel` module

This is defines the three signals that exist for each Antenna: `Ofdm`, `Nosie`, and `Ber`, and defines also what functions are needed to initialise them.

Listing D.15: The overall 'Channel' medium class: 'channel.h' header file.

```
/******************************************************************************
 * Sarah A Hugo
 * #09-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 ******************************************************************************
 * This code is freely available under the GNU General Public License.
 ******************************************************************************
 *  Channel.h
 *     -- header file for simulating a channel (aka a transmission medium)
 *        --> using *bitsstrm* to simulate modulation + noise
 *        --> bandwidth, frequency, noise, etc.
 *        --> relies on Ofdm and Noise classes
 *            --> in turn, based on BitsStrm and Binary
 *        --> call THIS instead of anything else
 ******************************************************************************
 * Because Channel is eventually going into a deque (in Mimo), its a struct,
 *   not a class. Do NOT make a class, or fear the wrath of the compiler. You
 *   have been warned. Keep it a struct, and leave memory management to the
 *   deque.
 *****************************************************************************/


#ifndef _CHANNEL_H_
#define _CHANNEL_H_

// for debuging
#include "debug.h"
// standard header -- i/o, types, exceptions, namespaces
#include "standard.h"
// for standard antenna/signal values
#include "allvals.h"
// the noise signal definition
#include "noise.h"
// the ofdm signal defintion
#include "ofdm.h"
// the ber signal defintion
#include "ber.h"


namespace Sim4G
{
    // further addition to the Sim4G:
    // the Channel class for properties of a transmission medium
    // call this to send signals back and forth between antennas, etc.
    struct Channel
    {
    private:
        // PRIVATE DATE MEMBERS
        // the size of strms (aka the bandwidth) requested
        size_t bandwidth;
        // a bitsstrm of modulated signal
        Ofdm* ofdmSig;
        // a noisy signal to add
        Noise* noisySig;
        // the difference between the signals that occurs during
        // transmission in the medium
        Ber* berSig;

        /* Accessors: keep private (for now)
         * access to actual signals is only done within the class
         */
        Ofdm* GetOfdmSig();
        Noise* GetNoiseSig();
        Ber* GetBerSig();
        // PRIVATE FUNCTIONS
        /* to create signals -- don't allow public access
         */
        void CreateASignalStrm(size_t sigType, bool isADebugRun);
        void MakeChannel(size_t, bool);
    public:
        /* creation of the channel (transmission medium)
```

```
         *   in this case thru air
         */
        Channel(size_t numSignalsNeeded, bool doIDebug,
                size_t newBand): bandwidth(newBand)
        {
            if (numSignalsNeeded <= 0) {
                return;
            } else {
                MakeChannel(numSignalsNeeded, doIDebug);
            }
        }

        /* copy constructer -- VITAL b/c its a *struct* not a class
         */
/*          Channel(const Channel& other)
        {
            Channel(C ;
        }
*/
        // OTHER FUNCTIONS
        Channel operator= (Channel& other);
        Channel operator*();
        // allow copying
        bool Healthy()
        {
            return (ofdmSig->IsSignalOk() &&
                    noisySig->IsSignalOk() && berSig->IsSignalOk());
        }

        // BER: how successful sending the signal was
        void FindBER();         // find the BER()
        void ChannelOutput();   // output BER to file / strings
        // send and receive signal (from 'unknown' antennas)
        void SendSignal();
        void SendSignalWithNoise();
        void SendSignalWithoutNoise();
        void ReceiveSignal();
        void ReceiveSignalWithNoise();
        void ReciveSignalWithoutNoise();
    };
}


// end of _CHANNEL_H_
#endif
```

And the implementation file. Close observers might note the very distinct similarity between the Signal's. This was, in fact, deliberate, and is highlighted and explained in Section D.10 to Section D.13.

Listing D.16: The overall 'Channel' medium class: 'channel.cpp' implementation file.

```
/*****************************************************************************
 * Sarah A Hugo
 * #09-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 *****************************************************************************
 * This code is freely available under the GNU General Public License.
 *****************************************************************************
 *  Channel.cpp
 *     -- source file for channel implemation
 *        --> all those extra details
 *****************************************************************************
 * Because Channel is eventually going into a deque (in Mimo), its a struct,
 *   not a class. Do NOT make a class, or fear the wrath of the compiler. You
 *   have been warned. Keep it a struct, and leave memory management to the
 *   deque.
 ****************************************************************************/


// my headers
#include "standard.h"
#include "channel.h"
#include "ofdm.h"
#include "noise.h"
```

```
#include "allvals.h"

using Sim4G::Debug;
using Sim4G::Channel;
using Sim4G::Ofdm;
using Sim4G::Noise;
using Sim4G::Ber;
using Sim4G::BitsStrm;
using Sim4G::MacsVals;

// PRIVATE FUNCTION
// create signal bitsstrms, as large as the requested bandwidth
void Channel::CreateASignalStrm(size_t thisSig, bool doIDebug)
{
    MacsVals macsStandards = MacsVals();
    size_t sigSize;
    BitsStrm bitsToAdd;
    std::string myStr;
    if (thisSig == macsStandards.ofdmType) {
        // creating ofdm BitsStrm
        Ofdm* tmp = new Ofdm(doIDebug, bitsToAdd, this->bandwidth);
        this->ofdmSig = tmp;
        sigSize = this->ofdmSig->SigSize();
        myStr = "OFDM␣";
    } else if (thisSig == macsStandards.noiseType) {
        // creating noisy BitsStrm
        Noise* tmp = new Noise(doIDebug, bitsToAdd, this->bandwidth);
        this->noisySig = tmp;
        sigSize = this->noisySig->SigSize();
        myStr = "Noise";
    } else if (thisSig == macsStandards.berType) {
        // set up BER channel: difference between signals at transmission
        // add many more until until our 'ber' is necessary size
        Ber* tmp = new Ber(doIDebug, bitsToAdd, this->bandwidth);
        this->berSig = tmp;
        sigSize = this->berSig->SigSize();
        myStr = "BER␣␣";
    }
    // signal created
    Debug(doIDebug) << "␣␣" << myStr << "␣channel␣aok,␣size␣=␣" << sigSize;
}


// PRIVATE FUNCTION
// helper to set up the channel
void Channel::MakeChannel(size_t numSignals, bool doIDebug)
{
    // create as many as needed
    MacsVals macsStandards = MacsVals();
    for (size_t i=1; i<=numSignals; i++) {
        Debug(doIDebug) << "␣␣" << i;
        CreateASignalStrm(macsStandards.ofdmType, doIDebug);
        CreateASignalStrm(macsStandards.noiseType, doIDebug);
        CreateASignalStrm(macsStandards.berType, doIDebug);
        // check it was ok
/*        try {
            if (this->GetOfdmSig()->IsSignalOk()) {
                Debug(doIDebug) << "    Channel(" << i
                    << "): displaying binary(s) in ofdm channel";
                size_t tmpNum = (this->GetOfdmSig()->GetSymbol(25));
                Debug(doIDebug) << "    " << tmpNum;
            }
        } catch(out_of_range) {
            Debug(doIDebug) << "    Chanel(" << i
                << "): could not be printed";
        }*/
        // start again at creating a signal
    }
}


// PRIVATE ACCESS
// so that access to actual streams is only done within the class
// overall access only knows about the channel
Ofdm* Channel::GetOfdmSig()
{
    return this->ofdmSig;
}
Noise* Channel::GetNoiseSig()
{
    return this->noisySig;
}
Ber* Channel::GetBerSig()
{
    return this->berSig;
}

// PUBLIC FUNCTION(S)

/* a little operator overloading:
 * assignment with another Channel passed by reference
 */
Channel Channel::operator= (Channel& other)
{
    // swap values
    this->ofdmSig = other.ofdmSig;
    this->noisySig = other.noisySig;
    this->berSig = other.berSig;
    this->bandwidth = other.bandwidth;
    return *this;
}
/* more overloading:
 * indirection: the * operator, aka, access to 'this'
 * VITAL for the copy constructor
 */
Channel Channel::operator*()
{
    return *this;
}
```

D.10 The Signal module

The Signal class, which is actually a virtual class. The virtual keyword means that classes which inherit off it can rewrite it as needed (to be explained later).

All signals share *basic characteristics*, such as amplitude, bandwidth, et cetera. However, the main focus of the program was always on a bit implementation, and the Signal classes (see Section D.11 to Section D.13) would be consistently using the provided bit implementation — the BitsStrm type mentioned here. It would only be how it was implemented that would vary. Moreover, each Signal implementation would consistently want to access the BitsStrm in certain ways, and require certain types of data. To save retyping code for each class, it was far simpler to create a virtual class, and simply inherit functionality and data types off the virtual class.

That was the aim of providing this virtualised. class.

Listing D.17: The abstract 'Signal' class: 'signal.h' header file.

```
/*******************************************************************************
 * Sarah A Hugo
 * #9-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 *******************************************************************************
 * This code is freely available under the GNU General Public License.
 *******************************************************************************
 *  Signal.h
 *     -- header file for the (abstract-ish) signal class
 *        --> defines basic signal characteristics.
 *******************************************************************************
 * Implements the BitsStrm class in an easy to use class, that can be easily
 *   inherited by modulating (ie OFDM) and noise modeling classes in turn.
 *   Saves reinventing the wheel for signal basics each time.
 ******************************************************************************/

#ifndef _ADT_SIGNAL_H_
#define _ADT_SIGNAL_H_

// for debuging
#include "debug.h"
// standard header -- i/o, types, exceptions, namespaces
#include "standard.h"
// the bitsstrm definition -- throws bits around in a strm
#include "bitsstrm.h"
// the binary things
#include "binary.h"

namespace Sim4G
{

    // the basic (abstract-ish) Signal class
    // inherits functions, capabilities, etc from BitsStrm container
    class Signal
    {
    private:
        BitsStrm sigstrm;
    protected:
        // data that can be inherited/initialised by subsequent classes
        // once-off func: set bitsstrm, according to given characteristics
        void SetSigStrm(BitsStrm someSignal)
        {
            sigstrm = someSignal;
            return;
        }
        // once-off func: set bitstrm -- set to a given size (zeroed values)
        void SetSigStrm(size_t goThisFar)
        {
            for(size_t i=0; i<goThisFar; i++) {
                sigstrm.PushFront(Binary());
            }
            return;
        }
    public:
        // in lieu of a contstructor...
        virtual void SetUpSignal()
        {
            // calls the two SetSigStrm functions
            // to be defined however derived classes wish
            return;
        }
        // option (for using Signal as its own type)...
        virtual void SetUpSignal(BitsStrm someSignal, size_t goThisFar)
        {
            SetSigStrm(someSignal);
            SetSigStrm(goThisFar);
            return;
        }
        // return sigstrm (bitsrream) so that derived classes...
        // can be call it / defined later as need arises
        BitsStrm GetSigStrm()
        {
            return sigstrm;
        }

        // check stream
        bool IsSignalOk()
        {
            return !(this->sigstrm.IsEmpty());
        }

        // get info about the strm --> can also be defined later (if need be)
        // -- defined here, tho, to provide option of saving time later
        deque<Binary>* GetAllStrm()
        {
            return this->sigstrm.GetBitStrm();
        }
        std::string GetOneString(size_t pos = 0)
        {
            return this->sigstrm.GetStrmString(pos);
        }
        std::string GetAllString()
        {
            this->sigstrm.GetStrmString(this->sigstrm.Begin(),
                this->sigstrm.End());
        }
        size_t GetOneNum(size_t pos = 0)
        {
            return this->sigstrm.GetStrmNum(pos);
        }
        deque<unsigned long> GetAllNum()
        {
            return this->sigstrm.GetStrmNum(this->sigstrm.Begin(),
                this->sigstrm.End());
        }

        // some basic signal-related functions
        // to be defined in subsequent classes as desired
        virtual BitsStrm Modulation()
        {
            return (sigstrm);
        }
        virtual BitsStrm Demodulation()
        {
            return (sigstrm);
        }

        // operator overloading
        Signal operator= (Signal* other)
        {
            this->sigstrm = other->sigstrm;
            return *this;
        }
    };

}

#endif
// end of _ADT_SIGNAL_H_
```

And the implementation file. As its a virtual class, all implementation is in the header file. As before though, the implementation file is included for correctness and to ensure compilation.

Listing D.18: The basic 'Signal' class: 'signal.cpp' implementation file.

```
/*****************************************************************************
 * Sarah A Hugo
 * #9-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 *****************************************************************************
 * This code is freely available under the GNU General Public License.
 *****************************************************************************
 *  Signal.h
 *     -- header file for the (abstract-ish) signal class
 *         --> defines basic signal characteristics.
 *****************************************************************************
 * Abstract class.
 * All definitions are in the header file. No need for anything here.
 ****************************************************************************/

#include "signal.h"
```

D.11 The Noise module

This is the first of three Signal class. It inherits off the Signal class, by the line

```
class Noise:  public Signal
```

This makes accessible to the class all the functions and types `Signal` defined in the same `public/private` way they were defined in `Signal`. In this way, `Noise` has access to `Signal`'s functions, has the option of rewriting these functions (to create its own implementation of a `Signal`). Moreover, when `Noise` is implemented in external classes as an object, it keeps its data type, the `BitsStrm` implementation, private, to avoid unwanted modification.

Listing D.19: The more advanced 'Noise' signal class: 'noise.h' header file.

```
/*****************************************************************************
 * Sarah A Hugo
 * #9-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 *****************************************************************************
 * This code is freely available under the GNU General Public
 *   License.
 *****************************************************************************
 *  Noise.cpp
 *      -- source file for Noise simulation
 *          --> echoes and fading effects on the bitsstrm via random numbers
 *          --> used to represent the signal
 *****************************************************************************
 * Implements the Signal (basic) class and gives the details of how it applies
 *   to noise (Rice and Rayleigh, *not* Gaussian).
 * As it expands on Signal, it also uses BitsStrm's.
 ****************************************************************************/

#ifndef _NOISE_SIG_H_
#define _NOISE_SIG_H_

// for debuging
#include "debug.h"
// standard header -- i/o, types, exceptions, namespaces
#include "standard.h"
// the abstract signal definition
#include "signal.h"
// the bitsstrm definition -- throws bits around in a strm
#include "bitsstrm.h"

namespace Sim4G
{

    // the Noise class:
    // an implementation of the (virtual/abstract) Signal class
    class Noise: public Signal
    {
    private:
        bool isADebugRun;
        size_t bandwidth;
        // protect default constructor
        Noise()
        { }
        // reimplementation and addition to the Signal class
        void SetUpSignal(BitsStrm);

    public:
        explicit Noise(bool doIDebug, BitsStrm& someStrm, size_t someNum):
                   bandwidth(someNum)
        {
            // for debugging purposes
            isADebugRun = doIDebug;
            // initialise the strms
            this->SetUpSignal(someStrm);
        }
        ~Noise()
        {    }

        // Get Signal
        Signal* GetSignal()
        {
        return this;
        }

        // sizes:
        // size of the "stream" associated to the signal
        size_t SigSize();
        // size of memory
        size_t MaxCapacity();

    };

}

// end of _NOISE_SIG_H_
#endif
```

And the implementation file. This is where `Noise` creates the randomised signal, and defines the extra details to its `Signal` model.

Listing D.20: The more advanced 'Noise' signal class: 'noise.cpp'implementation file.

```
/******************************************************************************
 * Sarah A Hugo
 * #9-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 ******************************************************************************
 * This code is freely available under the GNU General Public License.
 ******************************************************************************
 *  Noise.cpp
 *     -- source file for noisy signal implementations
 *        --> details of fading, echoes, etc.
 ******************************************************************************
 * Implements the Signal (basic) class and gives the details of how it applies
 *  to noise (Rice and Rayliegh, *not* Gaussian).
 * As it expands on Signal, it also uses BitsStrm's.
 *****************************************************************************/

// extra headers (for this file only)
#include <ctime>    // for the time() function
#include <cstdlib>  // for the srand() and rand() functions

// standard header includes
#include "standard.h"
#include "debug.h"

// my implementation files
#include "signal.h"
#include "noise.h"
#include "bitsstrm.h"
#include "binary.h"

using Sim4G::Debug;
using Sim4G::Signal;
using Sim4G::Noise;
using Sim4G::BitsStrm;
using Sim4G::Binary;


void Noise::SetUpSignal(BitsStrm somethingToAdd)
{
    // srand() and rand() are in cstdlib (see above)
    // and are used to generate random numbers (for noise)

    // randomize our 'pseudo-random' number generator
    // so its numbers are always different
    srand ( time(NULL) );
    // push random numbers into stream
    size_t goThisFar = (this->bandwidth - 1);
    for (size_t i=0; i<goThisFar; i++) {
        // generate random (bool) values and insert into somethingToAdd
        somethingToAdd.PushBack(Binary(rand() % 1));
    }

    // add randomized "stream" to our signal
    this->SetSigStrm(somethingToAdd);
}


// size(s): of the stream
size_t Noise::SigSize()
{
    return this->GetSigStrm().Size();
}
// size(s): of the memory
size_t Noise::MaxCapacity()
{
    return this->GetSigStrm().MaxCapacity();
}
```

D.12 The Ofdm module

Similar to Noise, Ofdm creates its own implementation of the Signal class, redefining functions as needed.

Listing D.21: The more advanced 'Ofdm' signal class: 'ofdm.h' header file.

```
/******************************************************************************
 * Sarah A Hugo
 * #9-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 ******************************************************************************
 * This code is freely available under the GNU General Public License.
 ******************************************************************************
 *  Ofdm.h
 *     -- header file for orthogonal frequency division multiplexing
 *        --> overall look at class
 *        --> simplified implementation of IFFTs and FFTs
 *            --> via modification of a bitsstrm
 ******************************************************************************
 * Provides the OFDM implemation of the Signal class, and the bit- based
 *  implementation (if time) of an IFFT and FFT.
 * As it expands on Signal, it also uses BitsStrm's.
 *****************************************************************************/

#ifndef _OFDM_SIG_H_
#define _OFDM_SIG_H_

#ifndef M_PI
#define M_PI 3.14159265359f
#endif

// for debuging
#include "debug.h"
// standard header -- i/o, types, exceptions, namespaces
#include "standard.h"
// the abstract signal definition
#include "signal.h"
// the bitsstrm definition -- throws bits around in a strm
#include "bitsstrm.h"


namespace Sim4G
{
    // the OFDM class:
    // an extension/implementation of the (virtual/abstract) Signal class
    class Ofdm: public Signal
    {
    private:
        // private data members
        bool isADebugRun;
        size_t bandwidth;
        // private implementation of the basic Signal functions
        void SetUpSignal(BitsStrm&);

        void ModulateSignal();        // simplified IFFT -- bitsstrm version
        void DeModulateSignal();      // simplified FFT  -- bitsstrm version

    public:
        // set up the signal system
        Ofdm(bool doIDebug, BitsStrm& someStrm, size_t someNum):
             bandwidth(someNum)
        {
            // for debugging purposes
            isADebugRun = doIDebug;
            // initialise the strms
            this->SetUpSignal(someStrm);
        }
        // stop the signal system
        ~Ofdm()
        {    }

        // check the signal
        size_t GetSymbol(size_t pos = 0)
        {
            return this->GetOneNum(pos);
        }
        // accessors
        // Get Signal
        Signal* GetSignal()
        {
            return this;
        }

        // sizes: of the "stream" associated with the signal
        size_t SigSize();
        // sizes: of the memory allocated to the "stream"/signal
        size_t MaxCapacity();

        // public implementation and addition to the Signal class

        // send the signal - output results of modulation
        Signal* SendSignal()
        {
            //ModulateSignal();
            return GetSignal();
        }
        // receive the signal - output results of demodulation
        Signal* ReceiveSignal()
        {
            //DeModulateSignal();
            return GetSignal();
        }

    };

}

#endif
// end of _OFDM_SIG_H_
```

And the implementation file. Note that in actually setting up the Signal, it required placing in it an actual signal (this is according to the basic signal model, mentioned in Section 2.4.1).

Listing D.22: The more advanced 'Ofdm' signal class: 'ofdm.cpp' implementation file.

```
/*******************************************************************************
 * Sarah A Hugo
 * #9-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 *******************************************************************************
 * This code is freely available under the GNU General Public License.
 *******************************************************************************
 *  Ofdm.cpp
 *     -- source file for orthogonal frequency division multiplexing
 *        --> extra details of implentation using Signal class.
 *        --> simplified implementation of IFFTs and FFTs
 *            --> via modification of a bitsstrm
 *******************************************************************************
 * Provides the OFDM implemation of the Signal class, and the bit-based
 *   implementation (if time) of an IFFT and FFT.
 * As it expands on Signal, it also uses BitsStrm's.
 ******************************************************************************/


// standard header files
#include "standard.h"
#include "debug.h"
// my implementation files
#include "signal.h"
#include "ofdm.h"


// access to classes
using Sim4G::Debug;
using Sim4G::Signal;
using Sim4G::Ofdm;
using Sim4G::BitsStrm;


/* generate a (basic) sinusoidal signal for later modulation and demodulation
```

```
 * and ‘push’ it onto the stream that represents our signal
 */
void Ofdm::SetUpSignal(BitsStrm& somethingToAdd)
{
    // make ’sinusoid’ and insert onto provided stream
    // Nyquist: sample at twice highest frequency present
    size_t goThisFar = (this->bandwidth)*2;
    for (size_t i=0; i<goThisFar; i++) {
        // max samples per second of a sinusoidal signal model
        somethingToAdd.PushBack( Binary((unsigned long)
                                 ( 100.0f*sinf(goThisFar*i*M_PI) )) );
    }
    // initialise the ofdm sig with generated sinusoidal "stream"
    this->SetSigStrm(somethingToAdd);
}

// size(s): of the stream
size_t Ofdm::SigSize()
{
    return this->GetSigStrm().Size();
}
// size(s): of the memory
size_t Ofdm::MaxCapacity()
{
    return this->GetSigStrm().MaxCapacity();
}
```

D.13 The Ber module

As before, Ber sets up its own version of a Signal implementation. In this case, it readies itself for finding the difference between Signals (note the FindTheDifference(Signal,Signal) function). This is where having a virtual Signal class comes in so handy. It does not matter to the Ber class what class they are — as long as they are the same *basic type*

Listing D.23: The more advanced 'Ber' signal class: 'ber.h' header file.

```
/******************************************************************************
 * Sarah A Hugo
 * #9-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 ******************************************************************************
 * This code is freely available under the GNU General Public License.
 ******************************************************************************
 *  Ber.cpp
 *     -- implementation file for BER (bit error rate)
 *        --> overall look at class
 *        --> combines a signal with noise to find the difference
 ******************************************************************************
 * Implements the Signal class (is of type Signal).
 * A signal, that is used to find the *difference* between two other signals.
 *****************************************************************************/


#ifndef _BER_SIG_H_
#define _BER_SIG_H_

// for debuging
#include "debug.h"
// standard header -- i/o, types, exceptions, namespaces
#include "standard.h"
// the abstract signal definition
#include "signal.h"
// the bitsstrm definition -- throws bits around in a strm
#include "bitsstrm.h"


namespace Sim4G
{
    /* The BER class:
     * an extension/implementation of the (virtual/abstract) Signal class
     * Sets up just like any other signal, but, given two signals, will
     * find the difference. (and store it)
     */
    class Ber: public Signal
    {
    private:
        // private data members
        bool isADebugRun;
        size_t bandwidth;
        // private implementation of the basic Signal functions
        void SetUpSignal(BitsStrm&);

    public:
        // set up the signal system
        Ber(bool doIDebug, BitsStrm& someStrm, size_t someNum):
            bandwidth(someNum)
        {
            // for debugging purposes
            isADebugRun = doIDebug;
            // initialise the strms
            this->SetUpSignal(someStrm);
        }
        // destruct
        ~Ber()
        {
        }

        // WARNING: may not be fully implemented.
        /* Finds the difference between two (given) signals
         * and stores it within itself.
         */
        void FindTheDifference(Signal*, Signal*);
        //---
        // Accessors
        // for later access
        Signal* GetSignal();
        // sizes:
        // size of the "stream" associated to the signal
        size_t SigSize();
        // size of memory
        size_t MaxCapacity();
    };
}

#endif
//end of _BER_SIG_H_ header
```

And the implementation file. Similar to 'Ofdm' and 'Noise', not much is in the implementation, as much of basic definition of signal characteristics has already been done in the 'Signal' class.

Listing D.24: The more advanced 'Ber' signal class: 'ber.cpp' implementation file.

```
/******************************************************************************
 * Sarah A Hugo
 * #9-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 ******************************************************************************
 * This code is freely available under the GNU General Public License.
 ******************************************************************************
 *  ber.cpp
 *     -- header file for BER (bit error rate)
 *        --> overall look at class
 *        --> combines a signal with noise to find the difference
 ******************************************************************************
 * Implements the Signal class.
 * A signal, that is used to find the *difference* between two other signals.
 *****************************************************************************/


// standard header files
#include "standard.h"
#include "debug.h"
// my implementation files
#include "signal.h"
#include "ber.h"

// access to classes
using Sim4G::Debug;
using Sim4G::Signal;
using Sim4G::Ber;
using Sim4G::BitsStrm;


/* SetUpSignal()
 * Make the BER signal reflect the given "stream" and our own bandwidth
 * passed down throughout the simulation *just* for this moment.
 */
void Ber::SetUpSignal(BitsStrm& somethingToAdd)
{
    // push vals onto stream, so its *not* empty
    size_t goThisFar = somethingToAdd.Size();
    for(size_t i=0; i<goThisFar; i++) {
        somethingToAdd.PushBack(1);
    }
    // make the BER signal real and large as requested
    this->SetSigStrm(somethingToAdd);
    this->SetSigStrm(this->bandwidth - 1);
}

//----
// PUBLIC FUNCTIONS

/* GetSignal()
 * Access to the actual BER signal
 */
```

```
Signal* Ber::GetSignal()
{
    return this;
}

/* size(s): of the stream
 */
size_t Ber::SigSize()
{
    return this->GetSigStrm().Size();
}
/* size(s): of the memory
 */
size_t Ber::MaxCapacity()
{
    return this->GetSigStrm().MaxCapacity();
}
```

D.14 The BitsStrm module

This module implements the Binary module, and turns it into a proper C++ “container”. That is, it provides it with iterators, and provides access to the container through the iterator.

Close observers may note that this is actually what in Java would be termed a “wrapper” class. That is, it places the object (Binary) in a container (the deque, pronounced ”deck”), and promptly calls it the deque a “stream” that is accessed through the deque’s own functions. However, as far as external modules know, the BitsStrm has a ”stream” inside it that it is accessing through iterators and stream-positions, just like any other high-level ‘container’ in C++. (The only thing it lacks to act as a true C++ “stream” is the over-riding the << and >> operators to act as extra (lazy) insertions and/or i/o ability...which were not done simply due to lack of time[4].)

Because it uses iterators, instead of the << and >> operators, however, focus shifts to the ‘pushing’ and ’popping’. The standard deque only comes the pop’s that don’t save, and push’s that only allow one type of variable to be inserted. This module focused on adding extra functionality, for all types of situations. (The implementation section will explain further.)

Listing D.25: The basic ’BitsStrm’ container: ’bitsstrm.h’ header file.

```
/*******************************************************************************
 * Sarah A Hugo
 * #9-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 ******************************************************************************
 * This code is freely available under the GNU General Public License.
 ******************************************************************************
 *  BitsStrm.h
 *     -- header file for bitsstrm (sends bits from A to B)
 *        --> overall look at class
 *        --> because C++ doesn't come with them automatically
 ******************************************************************************
 * Carries on from the Sim4G Binary module.
 * Implements what the <bitset> header in C++ fails to fully do: a binary
 *   object (decimal with binary rep.) in a container with iterators: allows
 *   access along the edges of the container -- a binary "stream" (aka a
 *   *deque* of binaries).
 * Also allows for basic math operations on "streams" (and on binary objects
 *   themselves) and for "stream" comparisions.
 *****************************************************************************/


#ifndef _BITSSTRM_H_MID_
#define _BITSSTRM_H_MID_

// for debuging
```

[4] And, in some cases, over-riding these operators can “break” the container. This was deemed a viable option when the module worked, and worked well, without this extra “functionality”.

```
#include "debug.h"
// standard header -- i/o, types, exceptions, namespaces
#include "standard.h"
// the binary definition -- the binaries everything is based on
#include "binary.h"



namespace Sim4G
{
    class BitsStrm
    {
    private:
        // define the "stream": binary numbers in a double-header container
        deque<Binary> binsStrm;
        //template<typename Strm> bs;

        // PRIVATE FUNC: protect unauthorised access to "stream"
        // Mimic the functionality of the deque for the "stream"
        // CAREFUL AND FINAL: removal from "stream"
        // do NOT use for i/o functions: suited to popping from "stream"
        void RemoveFromBack(Binary);            // saves one ptr
        void RemoveFromBack(size_t, BitsStrm&); // saves lots of pointers
        unsigned long RemoveFromBack(unsigned long&);  // returns saved num
        void RemoveFromBack(BitsStrm&);         // put into "stream"
        void RemoveFromFront(Binary);           // saves binary into pointer
        unsigned long RemoveFromFront(unsigned long&); // returns saved num
        void RemoveFromFront(size_t, BitsStrm&);// saves lots of pointers
        void RemoveFromFront(BitsStrm&);        // put into "stream"
    protected:
        // allowing for protected construction of "stream"
        //  (especially by inherited/derived classes)
        void MakeStrm()
        {
            // add to front of *empty* container (deque)
            Binary zeroedBinary = Binary();
            this->binsStrm.assign(1, zeroedBinary);
        }
        void MakeStrm(Binary binToAdd)
        {
            Binary zeroedBinary = Binary();
            this->binsStrm.assign(1, zeroedBinary);
            this->binsStrm.push_back(binToAdd);
        }
    public:
        // create bitsstrm
        BitsStrm()
        {
            MakeStrm();
        }
        BitsStrm(Binary binToAdd)
        {
            MakeStrm(binToAdd);
        }
        // destroy bitsstrm
        ~BitsStrm()
        {    }

        // "STREAM" INFORMATION
        // "stream" size
        size_t Size();
        // max capacity: size * size of binwords
        size_t MaxCapacity();
        // is it a empty bitsstrm?
        bool IsEmpty();
        void ClearStrm();
        Binary StrmAt(size_t);              // checked access
        Binary operator[](size_t) const;    // unchecked access


        // OP OVERLOADING

        // I/O OPS: rightshifting leftshifting (piping)

        // MATH OPS: arithmatic and assign
        BitsStrm operator= (BitsStrm& other);
        BitsStrm operator= (deque<Binary>&);

        // TRUTH OPS: boolean ops


        // access "stream" components
        // entire "stream"
        deque<Binary>* GetBitStrm()
        {
            return &binsStrm;
        }
        // get the binary numbers of the "stream":
        // "unchecked" access -- usually to first element in "stream"
        std::string GetStrmString();
        // checked access -- the string of the binary value at that strmpos
        std::string GetStrmString(size_t strmpos);
        // from first to last (begin to end)
        std::string GetStrmString(deque<Binary>::iterator,
                                  deque<Binary>::iterator);
        // get the numbers of the "stream":
        // "unchecked" access -- usually to first element in "stream"
        unsigned long GetStrmNum();
        // checked access -- equivalent decimal value at the given strmpos
        unsigned long GetStrmNum(size_t strmpos);
        // all the decimal values from where to where (usually begin to end)
        deque<unsigned long> GetStrmNum(deque<Binary>::iterator,
                                        deque<Binary>::iterator);
        // safer: single component of "stream"


        // "stream" iterators
        deque<Binary>::iterator Begin();
        deque<Binary>::const_iterator Begin() const;
        deque<Binary>::iterator End();
        deque<Binary>::const_iterator End() const;
        deque<Binary>::reference Front();
        deque<Binary>::const_reference Front() const;
        deque<Binary>::reference Back();
        deque<Binary>::const_reference Back() const;
        deque<Binary>::reverse_iterator RBegin();
        deque<Binary>::const_reverse_iterator RBegin() const;
        deque<Binary>::reverse_iterator REnd();
        deque<Binary>::const_reverse_iterator REnd() const;

        // insert operations
        deque<Binary>::iterator Insert(deque<Binary>::iterator, Binary&);
        void Insert(deque<Binary>::iterator, size_t, const Binary&);
        void Insert(deque<Binary>::iterator, deque<Binary>::iterator,
                    deque<Binary>::iterator);
        // assignment operations -- use with care!!
        void Assign(deque<Binary>::iterator, deque<Binary>::iterator);
        void Assign(size_t, Binary&);


        // popping -- saving what is 'popped' out of the "stream" if needed
        void PopBack();                     // standard
        void PopBack(size_t, BitsStrm&);    // save requested amount
        size_t PopBack(unsigned long&);     // save number, return it
        void PopBack(Binary);               // save into ptr
        void PopBack(BitsStrm&);            // save into "stream"
        void PopFront();                    // standard
        void PopFront(size_t, BitsStrm&);   // save requested amount
        size_t PopFront(unsigned long&);    // save number, return it
        void PopFront(Binary);              // save into ptr
        void PopFront(BitsStrm&);           // save into "stream"
        // pushing -- insertion of values into "stream"
        void PushBack(size_t);      // push this number of values
        void PushBack(Binary);      // push this binary onto "stream"
        void PushBack(BitsStrm&);   // append the "stream"
        void PushFront(size_t);     // insert onto front this many values
        void PushFront(Binary);     // insert this binary into front
        void PushFront(BitsStrm&);  // insert "stream" into front
    };
}

#endif
// end of _BITSSTRM_H_MID_
```

This, as always, is were the main work happens, and where all the functions are defined. As noted earlier, the main focus is on the 'push' and 'pop' functions — and in turn, on the customised 'insert' and 'remove'.

Although the extra functions ('insert' and 'assign') are provided, they are almost never used. It was found that once the stream was initialised, everything could be accomplished through the customised 'push' and 'pop' functions.

Listing D.26: The basic 'BitsStrm' container: 'bitsstrm.cpp' implementation file.

```
/*****************************************************************************
 * Sarah A Hugo
 * #9-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 *****************************************************************************
 * This code is freely available under the GNU General Public License.
 *****************************************************************************
 *  BitsStrm.cpp
 *     -- source file for bitsstrm (sending bits from A to B)
 *        --> extra details of class implemetation
 *        --> because C++ doesn't come with this automatically
 *****************************************************************************
 * Carries on from the Sim4G Binary module.
 * Implements what the <bitset> header in C++ fails to fully do: a binary
 *   object (decimal with binary rep.) in a container with iterators: allows
 *   access along the edges of the container -- a binary "stream" (aka a
 *   *deque* of binaries).
 * Also allows for basic math operations on "streams" (and on binary objects
 *   themselves) and for "stream" comparisions.
 ****************************************************************************/


// standard headers
#include "standard.h"
// my headers
#include "bitsstrm.h"
#include "binary.h"

// standard components
using std::deque;
// my own components
using Sim4G::BitsStrm;
using Sim4G::Binary;


//-------
// Mimic the behaviour of the deque
//-------

// Information about the "stream"/deque

// check size
size_t BitsStrm::Size()
{
    return this->binsStrm.size();
}
size_t BitsStrm::MaxCapacity()
{
    Binary tmpBin;
    this->PopFront(tmpBin);
    return (this->Size() * tmpBin.WordSize());
}
// check if its empty
bool BitsStrm::IsEmpty()
{
    return this->binsStrm.empty();
}
void BitsStrm::ClearStrm()
{
    return this->binsStrm.clear();
}
// checked access
Binary BitsStrm::StrmAt(size_t somePos)
```

```
{
    return this->binsStrm.at(somePos);
}
// unchecked access to bitsstrm -- so be careful
Binary BitsStrm::operator[](size_t somePos) const
{
    if (somePos < 0) {
        throw std::out_of_range("BitsStrm():Failed␣range␣check");
    } else {
        return this->binsStrm[somePos];
    }
}

//-------
// OPERATOR OVERLOADING
//-------

//-------
// MATH OPS: arithmatic and assign
BitsStrm BitsStrm::operator= (BitsStrm& other)
{
    this->binsStrm = other.binsStrm;
    return *this;
}
BitsStrm BitsStrm::operator= (deque<Binary>& otherStrm)
{
    this->binsStrm = otherStrm;
    return *this;
}

//-------
// TRUTH OPS: boolean ops





// Mimic deque insertion: Wrappers for the deque/"stream" insertion
// to be used by classes that have this as a member (object)
deque<Binary>::iterator BitsStrm::Insert(deque<Binary>::iterator here,
                                         Binary& binVal)
{
    return this->binsStrm.insert(here, binVal);
}
void BitsStrm::Insert(deque<Binary>::iterator here, size_t count,
                      const Binary& binVal)
{
    this->binsStrm.insert(here, count, binVal);
    return;
}
void BitsStrm::Insert(deque<Binary>::iterator here,
                      deque<Binary>::iterator first,
                      deque<Binary>::iterator last)
{
    this->binsStrm.insert(here, first, last);
    return;
}

// Assignment via iterators
void BitsStrm::Assign(deque<Binary>::iterator iterFirst,
                      deque<Binary>::iterator iterLast)
{
    this->binsStrm.assign(iterFirst, iterLast);
    return;
}
void BitsStrm::Assign(size_t count, Binary &binToAssign)
{
    this->binsStrm.assign(count, binToAssign);
    return;
}


// Get the binary numbers of the "stream"

// return the binary string -- unchecked access
// returns the binary number, usually of the first element in the "stream"
std::string BitsStrm::GetStrmString()
{
    return this->StrmAt(0).GetBinStr();
}
// return the binary string -- checked acces
// returns the binary number of the given strmpos
std::string BitsStrm::GetStrmString(size_t pos)
{
    return this->StrmAt(pos).GetBinStr();
}
```

```
// returns lots of binary strings -- checked access
// returns the binary numbers of the "stream"
// between the iterators (from where to where)
std::string BitsStrm::GetStrmString(deque<Binary>::iterator first,
                                    deque<Binary>::iterator last)
{
    // somethng to hold it, deliminate, and a place to begin
    std::string stringToHoldIt;
    std::string delim = "␣";
    deque<Binary>::iterator myIter = this->Begin();
    // iterate through the "stream"
    for (size_t strmpos=0; myIter<=last; strmpos++, myIter++) {
        // if strmpos is within the given range, add to the string
        if (myIter >= first) {
            // add to the string
            stringToHoldIt.append(this->StrmAt(strmpos).GetBinStr());
            if (myIter != last) {
                stringToHoldIt.append(delim);
            }
        }
    }
    return stringToHoldIt;
}


// Get the decimal values of the "stream"

// returns a single decimal values -- unchecked access
// returns the decimal number, usually of the first element in the "stream"
unsigned long BitsStrm::GetStrmNum()
{
    return this->StrmAt(0).GetDec();
}
// returns a single decimal value -- checked access
// returns the equivalent decimal value at the given strmpos
unsigned long BitsStrm::GetStrmNum(size_t pos)
{
    return this->StrmAt(pos).GetDec();
}
// from where to where (usually begin to end) -- checked access
// returns the (deque) of decimals for the "stream" between the iterators
deque<unsigned long> BitsStrm::GetStrmNum(deque<Binary>::iterator first,
                                          deque<Binary>::iterator last)
{
    // something to hold it and a place to begin
    std::deque<unsigned long> dequeOfNums;
    deque<Binary>::iterator myIter = this->Begin();
    // iterate
    for (size_t strmPos=0; myIter<=last; strmPos++, myIter++) {
        // if strmPos is within the given range, add to our dequeOfNumbers
        if (myIter >= first) {
            // add to the dequeOfNumbers
            dequeOfNums.insert(dequeOfNums.end(), this->GetStrmNum(strmPos));
        }
    }
    return dequeOfNums;
}




// "stream" iterators
deque<Binary>::iterator BitsStrm::Begin()
{
    return this->binsStrm.begin();
}
deque<Binary>::const_iterator BitsStrm::Begin() const
{
    return this->binsStrm.begin();
}
deque<Binary>::iterator BitsStrm::End()
{
    return this->binsStrm.end();
}
deque<Binary>::const_iterator BitsStrm::End() const
{
    return this->binsStrm.end();
}
deque<Binary>::reference BitsStrm::Front()
{
    return this->binsStrm.front();
}
deque<Binary>::const_reference BitsStrm::Front() const
```

```
{
    return this->binsStrm.front();
}
deque<Binary>::reference BitsStrm::Back()
{
    return this->binsStrm.back();
}
deque<Binary>::const_reference BitsStrm::Back() const
{
    return this->binsStrm.back();
}
deque<Binary>::reverse_iterator BitsStrm::RBegin()
{
    return this->binsStrm.rbegin();
}
deque<Binary>::const_reverse_iterator BitsStrm::RBegin() const
{
    return this->binsStrm.rbegin();
}
deque<Binary>::reverse_iterator BitsStrm::REnd()
{
    return this->binsStrm.rend();
}
deque<Binary>::const_reverse_iterator BitsStrm::REnd() const
{
    return this->binsStrm.rend();
}

//-------
// Popping and Pushing
//-------

// Pop: with options of passing arguments to save what's been popped

// Popping from the back
// standard pop -- no saving
void BitsStrm::PopBack()
{
    binsStrm.pop_back();
}
// pop a certain amount from provided "stream"
void BitsStrm::PopBack(size_t someAmount, BitsStrm& bitsToSave)
{
    RemoveFromBack((unsigned long)someAmount, bitsToSave);
    return;
}
// pop a single binary value
void BitsStrm::PopBack(Binary binToSave)
{
    RemoveFromBack(binToSave);
    return;
}
// pop a single binary number and save it into argument
size_t BitsStrm::PopBack(unsigned long& savedNum)
{
    return (size_t)RemoveFromBack(savedNum);
}

// pop from back into provided "stream" (as much as given "stream" will hold)
void BitsStrm::PopBack(BitsStrm& bitsToSave)
{
    RemoveFromBack(bitsToSave);
    return;
}
// Popping from the back
// standard pop from front of "stream" -- no saving
void BitsStrm::PopFront()
{
    binsStrm.pop_front();
}
// pop from back a certain amount into provided "stream"
void BitsStrm::PopFront(size_t someAmount, BitsStrm& bitsToSave)
{
    RemoveFromFront(someAmount, bitsToSave);
    return;
}
// pop a single binary number and save it into argument
size_t BitsStrm::PopFront(unsigned long& savedNum)
{
    return (size_t)RemoveFromBack(savedNum);
}
// pop from "stream" front a single binary value
void BitsStrm::PopFront(Binary binToSave)
```

```
{
    RemoveFromFront(binToSave);
    return;
}
// pop from front into provided "stream" (as much as given "stream" will hold)
void BitsStrm::PopFront(BitsStrm& bitsToSave)
{
    RemoveFromFront(bitsToSave);
    return;
}

// Push: standard insertion, aka insertion on request

// Pushing into the back
// push into back of "stream" as many values as requested (from given "stream")
void BitsStrm::PushBack(size_t numToAdd)
{
    // checked addition
    // only add to "stream" if number reasonable (in positive range)
    if (numToAdd > 0) {
        // for checked addition to the "stream"
        for (size_t i=0; i<numToAdd; i++) {
            this->PushBack(Binary());
        }
    }
    return;
}

// push into back of "stream" a single binary value
void BitsStrm::PushBack(Binary newBin)
{
    // checked addition
    // check for safety that given binary is a-ok
    if (!newBin.HasFailed()) {
        // only get to here if it is a-ok
        // if "stream" is unitialised
        if (this->IsEmpty()) {
            // literally assign memory to the "stream"
            this->Assign(1, newBin);
        } else {
            // otherwise safe to add, so add memory to it
            this->binsStrm.push_back(newBin);
        }
    }
    return;
}
/* push into back of "stream" the new "stream"
 */
void BitsStrm::PushBack(BitsStrm& bitsToAdd)
{
    // checked addition
    // only add if the "stream" exists to add with/to
    if (!bitsToAdd.IsEmpty())
    {
        // add onto end of our stream the new "stream"
        this->Insert(this->End(), bitsToAdd.Begin(), bitsToAdd.End());
    }
    return;
}
/* push into front of "stream" as many values as requested
 */
void BitsStrm::PushFront(size_t numToAdd)
{
    // checked addition
    // only add to stream if number is reasonable (positive range)
    if (numToAdd > 0) {
        return;
    } else {
        // now safe for checked addition to the "stream"
        for (size_t i=0; i<numToAdd; i++) {
            this->PushBack(Binary());
        }
        return;
    }
}
/* push into front of "stream" the value requested
 */
void BitsStrm::PushFront(Binary newBin)
{
    // checked addition
    // check something there to add
    if (!newBin.HasFailed()) {
        // only get here if the 'newBin' is a-ok
        // if our "stream" is unitialised
        if (this->IsEmpty()) {
            // literally assign memory to the "stream"
            this->Assign(1, newBin);
        } else {
            // otherwise it has memory, so we can add memory to it
            this->binsStrm.push_front(newBin);
        }
    }
    return;
}
/* push provided "stream" onto front of our "stream"
 */
void BitsStrm::PushFront(BitsStrm& bitsToAdd)
{
    // checked addition
    // check for safety that 'bitsToAdd' is initialised with values
    if (!bitsToAdd.IsEmpty()) {
        // only get here if 'bitsToAdd' is a-ok
        this->Insert(this->Begin(), bitsToAdd.Begin(), bitsToAdd.End());
    }
    return;
}


//-------
// Private functions
//-------

// Access "stream" Components -- addition, removal
// (also can be used for i/o functions)

/*-------
 * Remove elements from "stream"
 * removes a certain number of bits, saves into (given) "stream"
 */
void BitsStrm::RemoveFromBack(size_t numBitsToTake, BitsStrm& savingStrm)
{
    // find a max size we need to go to
    size_t tempSize = this->Size();
    // make sure temp "stream" to store the binary values is empty
    savingStrm.IsEmpty();
    // check number to add is reasonable
    if (numBitsToTake <= 0) {
        // if unreasonable request (can't add zero or neg amounts)
        return;             // did not remove any, return 0
    } else if (numBitsToTake >= tempSize) {
        // if not enough bits, empty what can (aka clear "stream")
        savingStrm = this->binsStrm;
        return;     // the entire "stream"
    } else {
        // safe, so remove amount desired
        for (size_t i=0; i<numBitsToTake; i++) {
            savingStrm.Insert(savingStrm.End(), this->binsStrm[i]);
            this->PopBack();    // adjust our "stream" to reflect removal
        }
        return; // return proof desired amount removed
    }
}
// pop a single (binary) value and return the information as a decimal
unsigned long BitsStrm::RemoveFromBack(unsigned long& numToSaveInto)
{
    Binary tmpBin = this->Back();
    this->PopBack();
    return tmpBin.GetDec();
}

// remove from "stream" a binary number
// BETTER -- saves the ptr
void BitsStrm::RemoveFromBack(Binary newBin)
{
    // no need to check for NULL in newBinPtr
    // simple removal of single binary value to provided pointer
    newBin = this->Back();   // save pointer
    this->PopBack();    // remove ptr from "stream"
    return;
}

// Remove from "stream" to "stream" -- takes all values
// BETTER -- takes care of ptr values within "stream"
void BitsStrm::RemoveFromBack(BitsStrm& savingStrm)
{
    //check for safety
    if (savingStrm.IsEmpty()) {
        return;
    } else {
        // safe to pop out of our "stream" and into provided "stream"
        size_t numToTake = savingStrm.Size();
        for (size_t i=0; i<numToTake; i++) {
            savingStrm.Insert(savingStrm.End(), this->binsStrm[i]);
            this->PopBack();    // adjust our "stream" to reflect removal
        }
        return;
    }
}

//-------
// Save lots of pointers
// returns a "stream"
void BitsStrm::RemoveFromFront(size_t numBitsToTake,
                                BitsStrm& savingStrm)
{
    // find a max size we need to go to
    size_t tempSize = this->Size();
    // make sure temp "stream" to store the binary values is empty
    savingStrm.ClearStrm();
    // check number to add is reasonable
    if (numBitsToTake <= 0) {
        // if unreasonable request (can't add zero or neg amounts)
        return;             // did not remove any, return 0
    } else if (numBitsToTake >= tempSize) {
        // if too many bits around, empty what can
        savingStrm = this->binsStrm;     // aka the entire "stream"
        return;
    } else {
        // remove amount desired
        for (size_t i=0; i<numBitsToTake; i++) {
        savingStrm.Insert(savingStrm.End(), this->binsStrm[i]);
        this->PopBack();    // adjust our "stream" to reflect removal
        }
        return; // return proof desired amount removed
    }
}
// pop a single (binary) value and return the information as a decimal
unsigned long BitsStrm::RemoveFromFront(unsigned long& numToSaveInto)
{
    Binary tmpBin = this->Front();
    this->PopFront();
    return tmpBin.GetDec();
}
// pop and save the information from the "stream" into the single binary
void BitsStrm::RemoveFromFront(Binary newBin)
{
    // simple change of where pointers point to
    newBin = this->Front();  // save pointer and object
    this->PopFront();
    return;
}

// instead of popping and losing information, pop into provided "stream"
void BitsStrm::RemoveFromFront(BitsStrm& savingStrm)
{
    // check not given dud "stream"
    if (savingStrm.IsEmpty()) {
        return;
    } else {
        // safe to pop out of our "stream" and into provided "stream"
        size_t numToTake = savingStrm.Size();
        for (size_t i=0; i<numToTake; i++) {
        savingStrm.Insert(savingStrm.End(), 1, this->binsStrm[i]);
        this->PopFront();   // adjust our "stream" to reflect removal
        }
        return;
    }
}
```

D.15 The Binary module

This is truly the backbone of the simulation. On this module, is everything based. The focus is on storing a decimal (base-10) and its binary equivalent (base-2). To save space, and to hold true to the bit nature, the binary is stored with a number of bools, this is, true or false values that are on all computers 1 bit long. The advantage of using bools is that its far easier to check the value is assigned than if the other 1-bit value (char) was used, and the bools are themselves represented internally as 1 for true and 0 for false. However, in assigning how long the binary word is, it is easier use the size of the char, as this is an easier accessed standard definition than the bool.

Listing D.27: The basic ’Binary’ data: ’binary.h’ header file.

```
/*****************************************************************************
 * Sarah A Hugo
 * #9-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 ****************************************************************************
 * This code is freely available under the GNU General Public License.
 ****************************************************************************
 *  Binary.h
 *      -- header file for binary values
 *          --> overall look at class
 *          --> as C++ doesn’t come with them automatically
 *                the way I need to use them
 ****************************************************************************
 * Turns a binary number itno a C++ base object. The only thing it lacks as a
 *    "proper" C++ container is the use of iterators.
 ****************************************************************************/

#ifndef _BINARY_H_BASE_
#define _BINARY_H_BASE_

// for CHAR_BIT definition
#include <limits.h>

// for debuging
#include "debug.h"
// standard header -- i/o, types, exceptions, namespaces
#include "standard.h"

namespace Sim4G
{
    // structure to hold the Binary:
    // a decimal and its equivalent binary value
    struct Binary
    {
    private:
        // data members
        unsigned long dec;
        // positve characters, to display the binary version of the number
        // (only as long as the size of our ’dec’)
        vector<bool> bin; // true is 1, false is 0 :)

        // how to pack bits into each binary word
        enum
        {
            // determined at runtime according to particular
            // compilar and computer being run on
            BINWORDSIZE = (size_t)(sizeof(unsigned long)),
            BITSPERWORD = (size_t)(CHAR_BIT*sizeof(unsigned long)),
            MAXWORDSIZE = BINWORDSIZE*BITSPERWORD
        };
        // long int, all positive (as must be for binary numbers)
         // bin to dec, dec to bin, with/without parameters
        void DecToBin();
        void DecToBin(unsigned long somedec);
        unsigned long BinToDec();
        unsigned long BinToDec(Binary someBinChars);
        unsigned long BinToDec(std::string someBinStr);
    public:
        // pretend constructors
        Binary()
        {
            dec = 0;
            DecToBin();
        }
        Binary(unsigned long newDec)
        {
            dec = newDec;
            DecToBin();
        }
        // allow for straight access to decimal value
        unsigned long GetDec()
        {
            return dec;
        }
        // tweak access to binary: return the string of the binary value
        std::string GetBinStr();

        // STUCT RELATED FUNCTIONS
        // in lieu of a constructor, set the values manually
        void SetBin();
        void SetBin(unsigned long newDec);
        // size of binary 'word'
        unsigned long WordSize();
        // reset to the 'word' to zero
        void Reset();
        // are all of the binary values set to true? (computationaly intensive)
        bool All();
        // are any of the binary values set to true? (not so intensive)
        bool Any();
        // return true if there's no binary value to handle
        bool HasFailed();

        //...
        //...
        //...
        //...

        // BIT OPS: use the bit operations to do things

        // bitwise rightshift: rightshift with binaries
        Binary RShift (const Binary& other);
        // bitwise rightshift: rightshift with unsigned long
        Binary RShift (const unsigned long someval);
        // bitwise leftshift: leftshift with binaries
        Binary LShift (const Binary& other);
        // bitwise leftshift: leftshift with unsigned long
        Binary LShift (const unsigned long someval);
        // bitwise OnesComplement: NOT in general
        Binary OnesComp();
        // bitwise AND: ANDing with binaries
        Binary AND (const Binary& other);
        // bitwise AND: ANDing with unsigned longs
        Binary AND (const unsigned long someval);
        // bitwise OR: OR's with binaries
        Binary OR (const Binary& other);
        // bitwise OR: OR's with unsigned longs
        Binary operator| (const unsigned long someval);
        // bitwise XOR: XOR's with binaries
        Binary XOR (const Binary& other);
        // bitwise XOR: XOR's with unsigned longs
        Binary XOR (const unsigned long someval);

        // MATH OPS: assignment and arithmatic

        // assignment of a unsigned long value only
        Binary operator= (const unsigned long decvalue);
        // addition: addition of binaries
        Binary operator+ (const Binary& other);
        // addition: addition of decimals
        Binary operator+ (const unsigned long someval);
        // addition: addition with equals sign
        Binary operator+= (const Binary& other);
        // addition: addition with increments
        /*Binary operator++();
        Binary operator++();*/
        // subtraction: subtraction of binaries
        Binary operator- (const Binary& other);
        // subtraction: subtraction of decimals
        Binary operator- (const unsigned long someval);
        // subtraction: subtraction with equals sign
        Binary operator-= (const Binary& other);
        // subtraction: subtraction with decrements
        /*Binary operator--();
        Binary operator--();*/
        // multiplication: multiplication of binaries
        Binary operator* (const Binary& other);
        // multiplication: multiplication of decimals
        Binary operator* (const unsigned long someval);
        // multiplication: multiplication with equals sign
        Binary operator*= (const Binary& other);
        // division: division of binaries
        Binary operator/ (const Binary& other);
        // division: division of decimals
        Binary operator/ (const unsigned long someval);
        // division: division with equals sign
        Binary operator/= (const Binary& other);

        // TRUTH OPS: booleans

        // equality test -- with binary
        bool operator== (const Binary& other);
        // equality test -- with decimal
        bool operator== (const unsigned long someval);
        // notequal test -- with binary
        bool operator!= (const Binary& other);
        // notequal test -- with decimal
        bool operator!= (const unsigned long someval);


        // FRIENDLY OPS:
        // ...


        // end of struct::Binary

    };
}

// end of _BINARY_H_BASE_
#endif
```

This is where the 'backbone' is implemented, and where the decimal to binary (and vice versa) conversion actually happens. This section took the most thought to ensure the binary was an accurate representation of the decimal. Thankfully, C/C++ comes with modulus (`%`) operator and bool-assignment (`?`), that took care of most of the work. The syntax of the `?` operator is as follows:

```
var = exp-to-check ?  val-if-true :  val-if-false;
```

It is this operator usage, in `DecToBin()`, that is the focal-point of the entire module.

Listing D.28: The basic 'Binary' representation: 'binary.cpp' implementation file.

```
/*****************************************************************************
 * Sarah A Hugo
 * #9-042: Experimental Analysis of a 4G Mobile Network
 * ENG 4903 USQ Research Project
 *****************************************************************************
 * This code is freely available under the GNU General Public License.
 *****************************************************************************
 *  Binary.cc
 *      -- source file for binary values
 *          --> extra class definitions
 *          --> because C++ doesn't come with them automatically
 ******************************************************************************
 * Turns a binary number into a C++ base object. The only thing it lacks as a
 *    "proper" C++ container is the use of iterators.
 *****************************************************************************/

// my headers
#include "binary.h"
#include "standard.h"

// using namespace declarations
using std::string;
using std::vector;
// Sim4G
using Sim4G::Binary;


// Struct related functions:
// retreiving information about the function for the user

// STUCT RELATED FUNCTIONS
// size of binary 'word'
unsigned long Binary::WordSize()
{
    return MAXWORDSIZE;
}
// set the decimal value
void Binary::SetBin()
{
    this->dec = 0;
    DecToBin(0);
}
void Binary::SetBin(unsigned long newDec)
{
    this->dec = newDec;
    DecToBin(newDec);
}

// reset to the 'word' to zero
void Binary::Reset()
{
    this->dec = 0;
    DecToBin(0);
}

// return true if there's no binary value to handle
bool Binary::HasFailed()
{
    return bin.empty();
}

// are any of the binary values set to true?
// (computationally intensive)
bool Binary::All()
{
    // ...
    return false;
}

// are any of the binary values set to true?
// (not so intensive, but not as thorough)
bool Binary::Any()
{
    // ...
    return true;
}


Binary Binary::RShift (const Binary& other)
{
    return Binary(this->dec << other.dec);
}
// bitwise rightshift: rightshift with unsigned long
Binary Binary::RShift (const unsigned long someval)
{
    return Binary(this->dec << someval);
}
// bitwise leftshift: leftshift with binaries
Binary Binary::LShift (const Binary& other)
{
    return Binary(this->dec >> other.dec);
}
// bitwise leftshift: leftshift with unsigned long
```

```cpp
Binary Binary::LShift (const unsigned long someval)
{
    return Binary(this->dec >> someval);
}
// bitwise OnesComplement: NOT in general
Binary Binary::OnesComp()
{
    return Binary(~(this->dec));
}
// bitwise AND: ANDing with binaries
Binary Binary::AND (const Binary& other)
{
    return Binary(this->dec & other.dec);
}
// bitwise AND: ANDing with unsigned longs
Binary Binary::AND (const unsigned long someval)
{
    return Binary(this->dec & someval);
}
// bitwise OR: OR's with binaries
Binary Binary::OR (const Binary& other)
{
    return Binary(this->dec | other.dec);
}
// bitwise OR: OR's with unsigned longs
Binary Binary::operator| (const unsigned long someval)
{
    return Binary(this->dec | someval);
}
// bitwise XOR: XOR's with binaries
Binary Binary::XOR (const Binary& other)
{
    return Binary(this->dec ^ other.dec);
}
// bitwise XOR: XOR's with unsigned longs
Binary Binary::XOR (const unsigned long someval)
{
    return Binary(this->dec ^ someval);
}

// MATH OPS: assignment and arithmatic

// assignment of a unsigned long value only
Binary Binary::operator= (const unsigned long decvalue)
{
    this->dec = decvalue;
    BinToDec(this->dec);
    return *this;
}
// addition: addition of binaries
Binary Binary::operator+ (const Binary& other)
{
    return Binary(this->dec + other.dec);
}
// addition: addition of decimals
Binary Binary::operator+ (const unsigned long someval)
{
    return Binary(this->dec + someval);
}
// addition: addition with equals sign
Binary Binary::operator+= (const Binary& other)
{
    return Binary(this->dec + other.dec);
}
// addition: addition with increments
/*Binary Binary::operator++ {
    // postincrement
    this.dec++;
    DecToBin(this.dec);
    return *this;
}
Binary Binary::operator++ {
    // preincrement
    ++this.dec;
    DecToBin(this.dec);
    return *this;
}*/
// subtraction: subtraction of binaries
Binary Binary::operator- (const Binary& other)
{
    return Binary(this->dec - other.dec);
}
```

```
// subtraction: subtraction of decimals
Binary Binary::operator- (const unsigned long someval)
{
    return Binary(this->dec - someval);
}
// subtraction: subtraction with equals sign
Binary Binary::operator-= (const Binary& other)
{
    return Binary(this->dec - other.dec);
}
// subtraction: subtraction with decrements
/*Binary Binary::operator-- {
    this.dec--;
    DecToBin(this.dec);
    return *this;
}
Binary Binary::operator-- {
    --this.dec;
    DecToBin(this.dec);
    return *this;
}*/
// multiplication: multiplication of binaries
Binary Binary::operator* (const Binary& other)
{
    return Binary(this->dec * other.dec);
}
// multiplication: multiplication of decimals
Binary Binary::operator* (const unsigned long someval)
{
    return Binary(this->dec * someval);
}
// multiplication: multiplication with equals sign
Binary Binary::operator*= (const Binary& other)
{
    return Binary(this->dec * other.dec);
}
// division: division of binaries
Binary Binary::operator/ (const Binary& other)
{
    unsigned long temp;
    // avoid division by zero errors by simply returning zero
    if (other.dec==0) {
        return Binary(); // zeroed binary number
    } else {
        // otherwise return the division result
        temp = (unsigned long)(this->dec / other.dec);
        return Binary(temp);
    }
}
// division: division of decimals
Binary Binary::operator/ (const unsigned long someval)
{
    return Binary(this->dec / someval);
}
// division: division with equals sign
Binary Binary::operator/= (const Binary& other)
{
    return Binary(this->dec / other.dec);
}

// TRUTH OPS: booleans

// equality test -- with binary
bool Binary::operator== (const Binary& other)
{
    return (this->dec == other.dec);
}
// equality test -- with decimal
bool Binary::operator== (const unsigned long someval)
{
    return (this->dec == someval);
}
// notequal test -- with binary
bool Binary::operator!= (const Binary& other)
{
    return (this->dec != other.dec);
}
// notequal test -- with decimal
bool Binary::operator!= (const unsigned long someval)
{
    return (this->dec != someval);
}
```

```

// ---- PRIVATE FUNCS
// For Decimal to Binary conversion, and vice versa

/* this.dec has the value of the decimal to be converted to the
 * ’binary’ character array
 */
void Binary::DecToBin()
{
    // iterate thru the array, checking for modulos
    unsigned long mask = 1;
    // if intel: mask = (double) 1 << (BINWORDSIZE - 1);
    for (unsigned long i=0; i<MAXWORDSIZE; i++) {
        bin.push_back(this->dec & mask ? true : false);
    }
    return;
}

/* If someone requests the binary of a decimal they’ve provided
 */
void Binary::DecToBin(unsigned long somenum)
{
    this->dec = somenum;
    DecToBin();
    return;
}

/* If someone requests the decimal without details just return our
 * own decimal without fussing around
 */
unsigned long Binary::BinToDec()
{
    return this->dec;
}

/* If someone requests a strange decimal
 * make it our own decimal and return it
 */
unsigned long Binary::BinToDec(Binary somebin)
{
    if (somebin.dec != this->dec) {
        // begin the conversion process
        this->dec = somebin.dec;
        DecToBin(somebin.dec);
        return this->dec;
    } else {
        // otherwise return our own decimal ’cause it was the same
        return this->dec;
    }
}

// Converting given binary string to an actual decimal number
// (provided binary string is as long as we need)
unsigned long Binary::BinToDec(std::string someBinStr)
{
    // check string is long enough to fit
    size_t strSize = someBinStr.size();
    if (strSize < MAXWORDSIZE) {
        // nasty fix: because string resize will fill with 1 character only,
                // and I want to fill with multiple values
        std::string tmpStr = someBinStr;
        for ( ; strSize < MAXWORDSIZE; ) {
            // so, to make string longer, as much as needed
            // and fill with our ’temp’ string
            someBinStr.append(tmpStr);
            strSize = someBinStr.size();
        }
        // catch-all statement for greater strings than desired
        someBinStr.resize(MAXWORDSIZE);
        strSize = someBinStr.size();
    } else if (strSize > MAXWORDSIZE) {
        // if string not too long
        // crop string to suit (will have to accept loss of values)
        someBinStr.resize(MAXWORDSIZE);
        strSize = someBinStr.size();
    }
    // the string is now the size of MaxWordSize
    // knowing this, it can be inserted directly into our own values
    int newNum=0;
    std::string::reverse_iterator strRItr = someBinStr.rend();
    for (unsigned int i = 0; i<someBinStr.size(); i++, strRItr++) {
        // iterate through binary string (back-to-front) comparing each
        // character to ’1’ so know what we need to do
        if (*strRItr == ’1’) {
            // transfer to our bin, and add to decimal
            this->bin.insert(this->bin.begin(), true);
            newNum += (i*2);
        } else {
            // transfer to bin
            this->bin.insert(this->bin.begin(), false);
        }
    }
    // equivalent of new binary is in newNum, so save values and return
    this->dec = newNum;
    return this->dec;
}

// convert the char array to string for easier C++ input/output
std::string Binary::GetBinStr()
{
    std::string mystr;
    // check initialisation (empty string, not rubbish values)
    if (!mystr.empty()) {
        // *not* empty (something in it), initialisation wrong, so clear it
        mystr.clear();
    }
    // empty string, so its an easy insert (just one value at a time)
    for (size_t i=0; i<MAXWORDSIZE; i++) {
        // insert equivalent of the bool value into string as a character
        if (bin[i]) {
            mystr.append(1, ’1’);
        } else {
            mystr.append(1, ’0’);
        }
    }
    return mystr;
}
```