Software Infrastructure for NeXtRAD Development in Julia Programming Language

Stephanie Cavale Jonkers

A dissertation submitted to the Department of Electrical Engineering, University of Cape Town, in partial fulfilment of the requirements for the degree of Master of Science in Engineering.

Cape Town, November 2016

University of Cape Town

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the degree of Master of Science in Engineering in the University of Cape Town. It has not been submitted before for any degree or examination in any other university.

Signature of Author .....Signed by candidate.....

Signature removed

Cape Town
November 2016

Abstract

This dissertation presents the implementation of signal processing infrastructure in Julia Programming Language. The aim is to aid sea clutter analysis using NetRAD and NeXtRAD data. Scripts written in Julia Programming Language and supporting documentation on how to navigate through compressed HDF5 files, apply pulse compression, pulse-Doppler processing and an adaptive LMS filter for interference suppression is presented. Both serial and multi-core pulse compression and pulse-Doppler processing functions are implemented. The assessment of the algorithm computation times highlights Julia's dependence on large amounts of RAM and slow data movement between worker processes. Multi-core pulse compression on 130 000 pulses each with 2 048 samples was not found to be faster than the serial implementation. Multi-core pulse-Doppler processing was able to achieve a speedup of 1.6 for a dataset with 102 400 pulses. Datasets larger than 102 400 pulses resulted in a memory bottleneck. The adaptive LMS filter was validated by applying an OS-CFAR detector to match filtered data before and after filtering. The filter was unable to improve the precision or recall for highly cluttered pulses, but was able to reduce the number of highly cluttered pulses.

Acknowledgements

I would like to thank Dr Richard Lord for taking the time to explain his work on radio interference suppression, Dr Craig Tong for his willingness to help sort out my harddrive issues and all my colleagues at the RRSG Radar Lab who were always willing to help with brainstorming, setting up equipment or sharing resources.

I would like to thank ARMSCOR and the CSIR for their financial assistance.

I am grateful to my supervisor Associate Professor Daniel W. O'Hagan for helping me stay focussed and pushing me to produce a dissertation I can be proud of.

I would like to acknowledge my family for being a constant source of encouragement and praise.

To my late father, who taught me that nothing worth having comes easy, thank you.

And last but definitely not least I want to make special mention of my dearest boyfriend Daniel Baumann, without whom I probably would not have made it this far. Daniel your optimistic and determined nature has always been something I admired. Thank you for motivating me every day, for helping me edit my dissertation and for your constant support.

Contents

Declaration ii

Abstract iii

Acknowledgements iv

Contents v

List of Figures viii

List of Tables x

Nomenclature xi

1 Introduction 1

1.1 Background 1
1.2 Motivation 2
1.3 Contributions 2
1.4 Purpose of Study 4
1.5 Scope and Limitations 4
1.6 Dissertation Outline 4

2 Literature Review 6

2.1 Multistatic Radar 6
2.1.1 NetRAD 7
2.1.2 NeXtRAD Database 8
2.2 Signal Processing Techniques 8

2.2.1 Pulse Compression 8
2.2.2 Doppler Processing 13
2.2.3 Radio Interference Suppression 17
2.3 General-Purpose CPU Computing 23
2.4 Julia Language for Radar Signal Processing 23

3 NetRAD/NeXtRAD Sea Clutter Database 25
3.1 HDF5 Data Model and File Format 26
3.1.1 Julia HDF5 Interface 27
3.1.2 Data Acquisition Software 28
3.2 Chapter Summary 29

4 Radar Data Preprocessing Techniques 30
4.1 Design Considerations 30
4.2 Pulse Compression 31
4.2.1 Serial Pulse Compression Software 32
4.2.2 Results and Discussion 34
4.3 Doppler Processing 36
4.3.1 Serial Pulse-Doppler Processing Software 37
4.3.2 Results and Discussion 37
4.4 Radio Frequency Interference 39
4.4.1 Interference Suppression 42
4.4.2 Results and Discussion 44
4.5 Chapter Summary 48

5 Julia Parallel Computing 51
5.1 Multi-core Pulse Compression Software 53
5.1.1 Multi-core Pulse Compression Code 55
5.1.2 Results and Discussion 57
5.2 Multi-core Doppler Processing 59
5.2.1 Multi-core Doppler Processing Code 60
5.2.2 Results and Discussion 61
5.3 Chapter Summary 64

6 Conclusions and Recommendations **66**

A Multistatic Radar **69**

B Digital Filters **72**

C Julia for Ne(X)tRad Manual **75**
C.1 Getting Started 75
C.2 Data Preparation 76
C.3 Quickplots 76
C.4 Saving to binary files 77

D Useful HDF5 Object Operations **79**

E Hardware Specifications **81**

Bibliography **82**

List of Figures

1.1 Overview of infrastructure software flow. 3

2.1 An illustration of the pulse width dilemma in radar detection. . . 9

2.2 Raw recorded NetRAD LFM up chirp signal with the offset removed. 10

2.3 Block diagram of an adaptive LMS filter taken from Lord and Inggs [1]. 21

3.1 HDF5 Primary object hierarchy. 26

3.2 Seven HDF5 object types. 27

4.1 Fast time-slow time pulse radar matrix 31

4.2 Serial pulse compression computation time as a function of number of pulses 35

4.3 An example of an RTI plot before and after pulse compression. The scene is of the ocean, the high intensity curves in (b) indicate waves moving toward the radar over time. 36

4.4 Serial Doppler processing computation time as a function of number of pulses 39

4.5 Doppler processed Quick-look RTI plot 40

4.6 Geometry of drone detection experiment 41

4.7 Figure showing Wi-Fi contaminated NetRAD data. The Wi-Fi appears as the lighter horizontal lines which effectively masks the underlying target returns. 41

4.8 Highly cluttered area of dataset 1 42

4.9 RFI dataset 1 (a) before and (b) after adaptive LMS filter interference suppression 46

4.10 RFI dataset 2 (a) before and (b) after adaptive LMS filter interference suppression 47

4.11 RFI dataset 3 (a) before and (b) after adaptive LMS filter interference suppression 48

5.1 A figure illustrating the overall flow of data in the serial pulse compress algorithm described in NR_PulseCompressSC.jl. 54

5.2 A figure illustrating the overall flow of data for a 2 core pulse compression algorithm described in NR_PulseCompressionMC.jl. . 54

5.3 Pulse Compression algorithm computation time as a function of pulses processed 58

5.4 Modified pulse compression algorithm computation time 59

5.5 A figure illustrating the overall flow of data in the serial Doppler processing algorithm described in NR_DopplerProcessingSC.jl. . . 60

5.6 A figure illustrating the overall flow of data for a 2 core Doppler processing algorithm described in NR_DopplerProcessingMC.jl. . . 61

5.7 Doppler processing algorithm computation comparison 63

5.8 Hybrid Doppler processing algorithm computation comparison . . 64

List of Tables

3.1 Table of NetRAD trials with summary of objective. 25

3.2 Table of available Julia scripts for HDF5 NetRAD data acquisition. 28

4.1 Table of available Julia scripts for serial pulse compression. 32

4.2 Table outlining variable memory usage in serial pulse compression routine. 34

4.3 Table of available Julia script for Doppler processing. 37

4.4 Table of available Julia scripts for interference suppression 42

4.5 Definition of Decision Rule for Detection 45

4.6 Summary of OS-CFAR results for dataset 1 with 3 targets 50

4.7 Summary of OS-CFAR results for dataset 2 with 3 targets 50

4.8 Summary of OS-CFAR results for dataset 3 with 3 targets 50

5.1 Table of available Julia scripts for Pulse Compression 55

5.2 Table of available Julia scripts for Doppler processing 60

C.1 Table of available Julia scripts for raw data processing 78

D.1 Table of basic operations performed on a file 79

D.2 Table of basic operations performed on a group 79

D.3 Basic operations performed on a link 80

E.1 Hardware specifications of the server used for computation 81

Nomenclature and Abbreviations

SNR	Signal-to-noise, the ratio of signal power to noise power.
Straddle Loss	The loss observed when the peak signal amplitude is not sampled.
DTFT	Discrete Time Fourier Transform, achieved by sampling a continuous time signal.
DFT	Discrete Fourier Transform, achieved by sampling the DTFT.
FFT	Fast Fourier Transform, a particular implementation of the DFT.
CPI	Coherent Processing Interval, a group of pulses that can be integrated coherently.
PRI	Pulse Repetition Interval, the reciprocal of the pulse repetition frequency which is the number of pulses transmitted per second.
FIR Filter	Finite Impulse Response Filter.
IIR Filter	Infinite Impulse Response Filter.
WLAN	Wireless Local Area Network.
LOS	Line-of-Sight.
RFI	Radio Frequency Interference.
VHF	Very High Frequency, (30-300) MHz.
UHF	Ultra High Frequency, (300-1000) MHz.
LMS	Least Mean Square.
MLE	Maximum Likelihood Estimation.
AR	Autoregression.
PML	Parametric Maximum Likelihood.
SAR	Synthetic Aperture Radar.
MTI	Moving Target Indication.
HDF5	Hierarchical Data Format 5.

Chapter 1

Introduction

This chapter introduces the major themes of the dissertation. First, the project is given context in Section 1.1 with a background of NetRAD. The need for software infrastructure for NetRAD and NeXtRAD is motivated in Section 1.2. Section 1.3 specifies the general computing and signal processing problems to be investigated. Section 1.4 formalises the purpose of the dissertation. Section 1.5 states the scope of the project and Section 1.6 gives an overview of the dissertation.

1.1 Background

In radar signal processing, sea clutter is a widely researched topic. Most research in this area, however, is based on monostatic radar data. As Section 2.1 will show, little research exists on the properties of sea clutter in a bistatic or multistatic configuration, primarily due to the lack of available data. NetRAD is a netted radar project that started with the purpose of collecting sea clutter data to aid the research in this field.

1.2 Motivation

Given that NetRAD's main purpose was to support research, the radar datasets were made available to any interested parties. Each dataset contains not only the (130 000 pulses$\times$2 048 range samples) radar echo data, but also metadata such as radar operating parameters, environmental conditions and sea state information relevant to each experiment. During data collection campaigns, external factors led to some datasets being corrupted. This is typical of practical measurement campaigns. Software infrastructure to perform necessary processing on raw data is a valuable asset not only for NetRAD, but for its successor NeXtRAD, which is currently under development. The infrastructure proposed would provide a user with the ability to quickly and conveniently produce radar plots with enhanced signal-to-noise, produce pulse-Doppler plots and in cases where there is radio frequency interference, provide a suppression technique.

1.3 Contributions

This dissertation contributes improved usability and data preparation support for multistatic radar systems, NetRAD and NeXtRAD. Large amounts of data has been collected since 2009, however, due to the inconsistencies in data formatting and lack of archiving, only a small portion of the available data has actually been analysed. This project has developed and documented a collection of software scripts to read raw data as well as metadata from compressed data files, convert samples from raw ADC values to voltage, increase the signal-to-noise ratio (SNR) using pulse compression, apply pulse-Doppler processing, and apply an interference suppression filter in an open-source programming language. The aim of this work is to aid sea clutter analysis from the raw data acquisition stage to filtered data preparation. The radar processing techniques have been implemented in a new scientific language, Julia, which demonstrates its use in a radar application and compares its performance for a single core versus a multi-core implementation.

Figure 1.1 gives a brief overview of the different goals intended for the infrastructure to be developed.

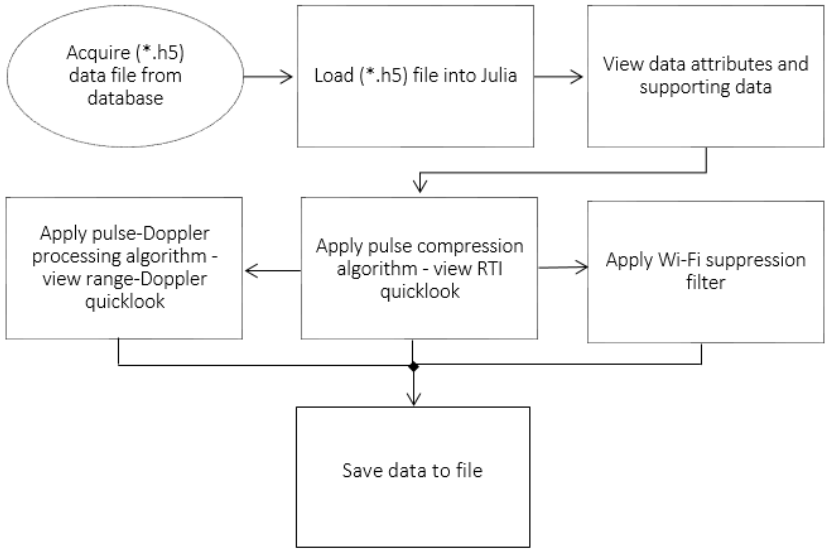


Figure 1.1: Overview of infrastructure software flow.

The main contributions can thus be summarised as:

1. Script to read NetRAD data from a binary/compressed HDF5/text file in Julia Programming Language.
2. Script to navigate through HDF5 file to extract radar datasets and metadata in Julia Programming Language.
3. Serial and multi-core script to produce pulse compressed data in Julia Programming Language.
4. Serial and multi-core script to produce pulse-Doppler processed data in Julia Programming Language.
5. Script to suppress radio frequency interference present in NetRAD data in Julia Programming Language.

1.4 Purpose of Study

This report forms part of an ongoing study on multi-frequency, multistatic radar systems. The amount of data produced in the NetRAD project is in the order of hundreds of Gigabytes and NeXtRAD its successor will only add to the large database. This study focuses on building software infrastructure in a new scientifically oriented language, to deal with large amounts of complex data and provide scientists and engineers with the tools to find relevant data, have quick access to appropriate statistical properties, and visualise the data to aid the analysis and future research.

1.5 Scope and Limitations

This report is limited to the development of pulse compression, pulse-Doppler processing and Wi-Fi interference suppression software in Julia Programming Language. It does not include software development in other languages. Although the software was developed specifically in support of the NetRAD and NeXtRAD projects initiated at University College London and University of Cape Town, it is open-source and can be used in any other application.

1.6 Dissertation Outline

The dissertation begins with a review on multistatic radar, radar signal processing techniques, multi-core computing, and an introduction to Julia Programming Language in Chapter 2. The advantages and disadvantages of multistatic radar systems are discussed in Section 2.1 followed by an overview of NetRAD and the database, which is still under development. When dealing with experimental recordings, it is necessary to process the data prior to subsequent analysis. Section 2.2 reviews the theory on pulse compression, Doppler processing and radio frequency interference suppression. Section 2.3 looks at parallel

computing architectures, which are used to speed up processing time. The chapter ends with an assessment of the suitability of Julia Programming Language in a radar signal processing context in Section 2.4.

Chapter 3 discusses the file format standard in which all NeXtRAD data will be stored. Section 3.1 explains the data model for HDF5 as well as the file format and explains how to use the Julia script to navigate through an HDF5 file to acquire relevant data. The chapter is summarised in Section 3.2.

Chapter 4 details the signal processing tools required for the preparation of NetRAD data. Section 4.2 describes the scripts developed for pulse compression and discusses the processed data and the computation time of the algorithm. Section 4.3 describes the script to implement pulse-Doppler processing, discusses the computation time as well as how to produce and interpret pulse-Doppler data visualisations. Section 4.4 describes a drone experiment using NetRAD and the WiFi contaminated datasets associated with it. The script which implements an adaptive least mean squares filter to suppress the interference is detailed, and the performance of the filter is assessed visually and is quantified by implementing an Ordered Statistic Constant False Alarm detector. This chapter is summarised in Section 4.5.

Chapter 5 explores Julia's multi-core platform and important themes related to parallel implementations. It introduces the concept of speedup and the degree of parallelism that can be expected from a multi-core algorithm. Sections 5.1 and 5.2 describe the parallel implementations of pulse compression and pulse-Doppler processing respectively. In each case the parallel implementation is compared to its serial counterpart in terms of computation time for various numbers of pulses considered. The chapter is summarised in Section 5.3.

Finally chapter 6 concludes the dissertation by reiterating the motivation, contributions and most important findings. Based on these results and findings recommendations for future work are made.

Chapter 2

Literature Review

2.1 Multistatic Radar

In literature, a collection of multiple monostatic radar or bistatic radar components which are separated spatially, have overlapping coverage area and work together as one system is loosely referred to as a multistatic, netted or multisite radar system. In many cases, however, the architecture of the system is not well defined. Target detection is usually done with each transmit-receive pair performing separate detections within the region of interest [2]. Derham, Doughty, Woodbridge and Baker [3] attempted to create a more well defined definition for a low-cost multistatic, netted radar system as one comprising of more than one transmitter and/or receiver, and in which complex signals are centrally processed to form a joint detection. In addition to this, all nodes are to have mutual temporal and spatial coherency. The spatial diversity gained by having multistatic radar systems allows for different aspects of a target to be viewed simultaneously. The NetRAD system comprises of one transmit-receive node and two receive-only nodes. In other words, there is one monostatic node and two bistatic nodes.

2.1.1 NetRAD

Due to the benefits of multistatic radar systems (See Appendix A for more details), they are becoming the favoured alternative as a more effective way to detect and track low-signature targets in cluttered environments. The NetRAD project was started to help produce data to aid research in this field.

NetRAD was a three node coherent multistatic netted radar system developed by University College London in 2000 [4, 5, 6]. Current literature on monostatic sea clutter is well researched [7, 8, 9, 10], but little data exists on bistatic and multistatic sea clutter [4]. As a result limited research on the topic exists. NetRAD was developed with the intention of providing data for understanding sea clutter and vessel properties in bistatic and multistatic configurations. The original system design was implemented as a cable synchronised, multi-node radar. Practically the 50 meter cables used to connect the nodes introduced limitations in the achievable bistatic angles as well as synchronisation. As a result, in 2003 the University of Cape Town joined the project to develop a centrally controlled wireless network, using a GPS disciplined synchronised set of clocks. This allowed nodes to be spaced much further apart and solved practical synchronisation problems. NetRAD operated in S-Band (the Wi-Fi Band) at 2.4 GHz, which meant that sites had to be carefully chosen. Nonetheless, data form some campaigns still exhibit Wi-Fi contamination [4, 6]. See Section 2.2.3 for Radio Frequency Interference Suppression.

NeXtRAD, the successor of NetRAD, currently still in development will be dual band, that is X- and L-Band [6]. It will be fully polarimetric in X-Band, i.e. co- and cross-polar components can be simultaneously received. There are two X-Band receive channels and one L-Band receive channel. It takes two cycles to gather the X-Band matrix and four cycles to gather the L-Band. In total it requires eight Pulse Repetition Intervals (PRIs) for both frequencies. The transmitter is to be switched between PRIs. Hybrid polarimetry is also planned.

2.1.2 NeXtRAD Database

A large amount of multistatic measurements of targets and clutter have been collected over the past years using the NetRAD system. At the time the chosen method of storing the data proved to be cumbersome and significantly hampered the usability of the data for analysis. It has therefore become necessary to organise the data into a database [5] such that the user can navigate and acquire data by specifying parameters such as the sea state, range and bistatic angle.

2.2 Signal Processing Techniques

This chapter reviews the fundamental theory of pulse compression, Doppler processing and WiFi interference suppression.

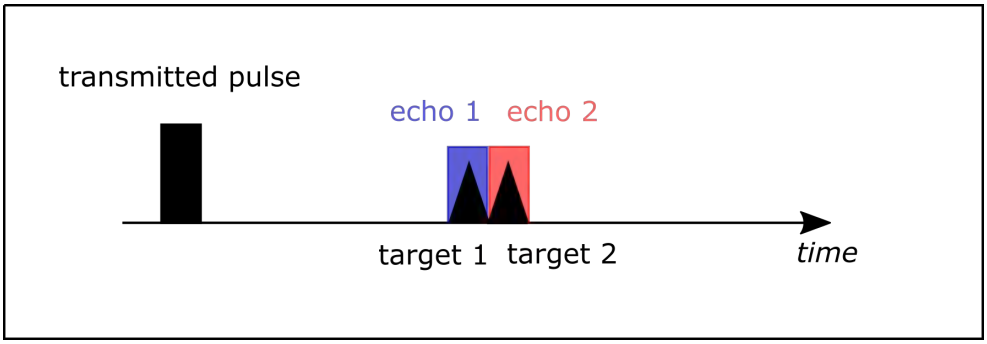
2.2.1 Pulse Compression

Pulse compression is a technique used in radar signal processing that tries to discriminate target signals from the initially dominant background noise by using a matched filter, which maximises the signal-to-noise ratio when the received signal weak relative to the transmitted signal [11, 12].

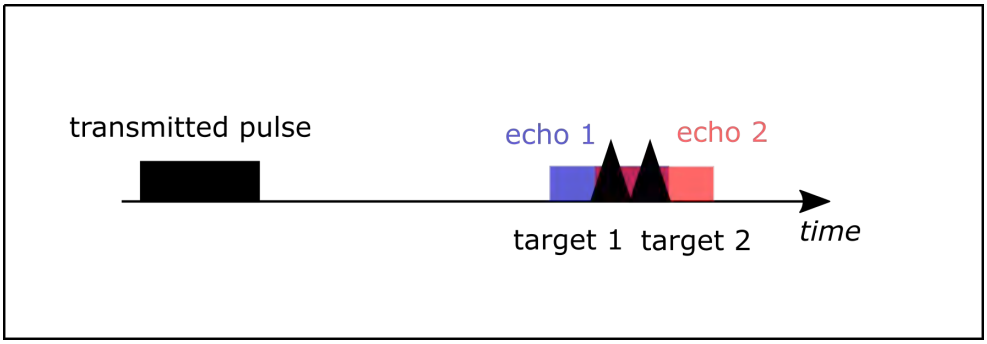
The need for pulse compression arises from the need to simultaneously have a pulse that has enough energy to travel large distances, while maintaining a high range resolution, which is achieved with a short pulse width [13]. Figure 2.1 illustrates the pulse width dilemma that pulse compression aims to solve. Ideally we would prefer the transmitted pulse to have a short pulse-width in order to have a good resolution of closely spaced targets, illustrated in sub-figure (a). At the same time, the maximum range that the signal can travel in space and still be reliably detected is proportional to the amount of energy in the signal. To achieve this, we would either need to increase the pulse width as shown in sub-figure (b) or increase the pulse amplitude. Unfortunately, there

are practical limits to the amount of peak power that can be produced, thus to increase the energy in the signal, we would require a longer pulse-width. These seemingly contradicting requirements can both be satisfied with the pulse compression technique when used together with specially designed waveforms [13].

Figure 2.1: An illustration of the pulse width dilemma in radar detection.



(a) A short transmitted pulse is able to discriminate between closely spaced targets, but there is limit to the amount of peak power and thus also the maximum detectable range.



(b) A long pulse has a larger maximum detectable range, but suffers from poor range resolution.

In theory, pulse compression can be performed in either the time domain or the frequency domain. Practically, to achieve a compressed signal output we convolve the sampled echo data in the time domain with a waveform (the matched filter), which is modulated in some way.

Waveforms In radar applications, the special waveform referred to is a time-reversed replica of the original transmitted signal. In NetRAD this waveform is a pulse which is linearly modulated in frequency. There are several other options for appropriate waveforms such as phase-coded waveforms and non-linear frequency modulated waveforms that could be used in pulse compression. The linear frequency modulated (LFM) waveform, however, is a popular choice in radar applications [13], [14, Ch.1] for its simplicity of implementation, its well established receiver filtering structure and its Doppler tolerance. Figure 2.2 shows the recorded reference signal transmitted and received on Node 3 with 500 samples.

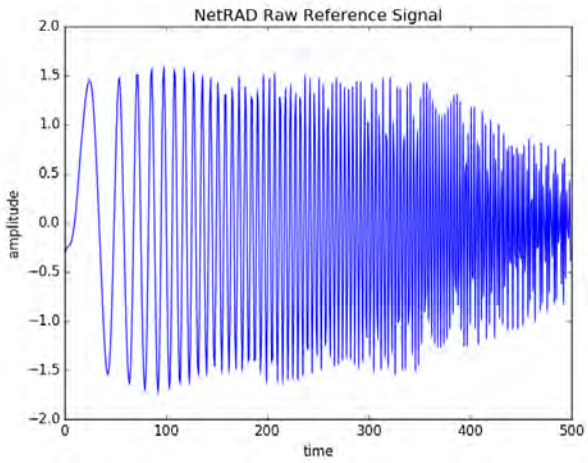


Figure 2.2: Raw recorded NetRAD LFM up chirp signal with the offset removed.

Notation: Vector Representation of Sampled Signals We introduce the notation also used in [14] for the vector representation of sampled signals, which will be used throughout this chapter. We will express the impulse response and

the signal as column vectors with the following notation respectively:

$$\mathbf{h} = [h[0] \;\; h[1] \;\; ... \;\; h[L-1]]^T, \qquad \mathbf{x} = [x[0] \;\; x[1] \;\; ... \;\; x[L-1]]^T. \tag{2.1}$$

Notation: Spectral Representation of Sampled Signals It is convenient to work in the frequency domain rather that in the time domain, for operations such as pulse compression since convolution in the time domain is multiplication in the frequency domain. Consider the impulse response of the system $h[n]$, then the z-transform, $H(z)$, relates the input and output in the z-domain via the convolution equation as follows

$$Y(z) = H(z)X(z). \tag{2.2}$$

Then since the Discrete Time Fourier Transform (DTFT) is just a z-transform evaluated at points on a unit circle, the DTFT of the impulse response is defined as *frequency response*. Mathematically,

$$Y(\hat{f}) = H(\hat{f})X(\hat{f}). \tag{2.3}$$

Where $\hat{f} = f/f_s$ is the normalised frequency and f_s is the original sampling frequency. The K-point DTFT frequency samples can then be found by

$$\hat{f} = \frac{k}{K}, \quad k \in \mathbb{Z}. \tag{2.4}$$

Matched Filters As has been mentioned before, in pulse compression the parameter which we try to maximise is the SNR. In the NetRAD project, we deal with a finite length filter, thus we will approach the problem of SNR optimisation by looking at the output of an FIR filter.

Consider a finite length signal $\mathbf{x}$ which has the signal of interest $\mathbf{s}$ plus noise $\mathbf{n}$. Mathematically,

$$\mathbf{x} = \mathbf{s} + \mathbf{n}. \tag{2.5}$$

The power of the output sample of an FIR filter signal $\left|y[n]\right|^2$ is[1]

$$|y|^2 = y^* y^T = \mathbf{h}^H \mathbf{x}^* \mathbf{x}^T \mathbf{h}. \tag{2.6}$$

here the superscript H is the Hermitian transpose. The objective now is to try and find filter coefficients in vector $\mathbf{h}$, such that the SNR is maximised. We start by looking at the power of the signal and that of the noise separately, they are defined as

$$|y|^2_{signal} = \mathbf{h}^H \mathbf{s}^* \mathbf{s}^T \mathbf{h} \qquad \text{and} \qquad |y|^2_{noise} = \mathbf{h}^H \mathbf{n}^* \mathbf{n}^T \mathbf{h}. \tag{2.7}$$

In radar signal processing theory the matrix formed from the product $\mathbf{R} = \mathbf{n}^* \mathbf{n}^T$ is also known as the covariance matrix. Now we can easily define the SNR as the ratio of the powers at the output of the filter

$$SNR = \frac{\mathbf{h}^H \mathbf{s}^* \mathbf{s}^T \mathbf{h}}{\mathbf{h}^H \mathbf{R}\ \mathbf{h}}. \tag{2.8}$$

Then to find the filter $\mathbf{h}$ that maximises the SNR, we make use of the Cauchy-Schwarz inequality, which states that for all vectors $\mathbf{p}$ and $\mathbf{q}$ of an inner-product space, it is true that

$$|\langle \mathbf{p}, \mathbf{q} \rangle|^2 \leq ||\mathbf{p}||^2 \cdot ||\mathbf{q}||^2. \tag{2.9}$$

Here $\langle\ ,\ \rangle$ is used to represent the inner product and $||\ .\ ||$ is the norm of the vector. The covariance matrix has the property that it is constant along each diagonal (i.e it is a Toeplitz matrix), and they have Hermitian symmetry meaning (i.e $\mathbf{R}^H = \mathbf{R}$) and are positive definite. If we then choose $\mathbf{p} = \mathbf{ah}$ and $\mathbf{q} = (\mathbf{a}^H)^{-1}\ \mathbf{s}^*$ we can define the covariance matrix $\mathbf{R}$ as

$$\mathbf{R} = \mathbf{a}^H \mathbf{a}. \tag{2.10}$$

[1]Note that the "$[n]$" notation is dropped for ease of reading.

This is now in the form required to apply the Cauchy-Schwarz inequality property namely,

$$\mathbf{h}^H\mathbf{s}^*\mathbf{s}^T\mathbf{h} \leq \left|\left|\mathbf{a}^H\mathbf{h}\right|\right|^2 \left|\left|(\mathbf{a}^H)^{-1}\mathbf{s}^*\right|\right|^2 = (\mathbf{h}^H\mathbf{R}\ \mathbf{h})(\mathbf{s}^T\mathbf{R}^{-1}\mathbf{s}^*). \tag{2.11}$$

Then dividing both sides by $\mathbf{h}^H\mathbf{R}\ \mathbf{h}$ we can recover an expression for the SNR and find that:

$$SNR = \frac{\mathbf{h}^H\mathbf{s}^*\mathbf{s}^T\mathbf{h}}{\mathbf{h}^H\mathbf{R}\ \mathbf{h}} \leq \mathbf{s}^T\mathbf{R}^{-1}\mathbf{s}^*. \tag{2.12}$$

And this is maximised when for some k, $\mathbf{aH} = k(\mathbf{a}^H)^{-1}\mathbf{s}^*$

$$\mathbf{H} = k\mathbf{R}^{-1}\mathbf{s}^*, \qquad k \in \mathbb{R}. \tag{2.13}$$

Digital Pulse Compression In digital pulse compression, at the final intermediate frequency, the complex envelope is recorded using an ADC with a technique known as digital-down conversion. Frequency-domain digital pulse compression processor is the method used in NetRAD, it is the preferred set-up since a time-domain convolution processor equivalent can be computationally intensive [11].

2.2.2 Doppler Processing

In Doppler processing we use the information in Doppler shift to gain insight into the target of interest. The more obvious use of the Doppler information is to calculate the radial velocity of the target. However, it is also particularly helpful for detecting targets in environments where the clutter is a dominant interference. Two main types of Doppler processing methods are discussed in this review, namely Moving Target Indication (MTI) and pulse-Doppler processing [15].

Doppler shift, which is observed as the change in radar frequency, is approximated

by the following relationship:

$$f_d = \frac{2v}{c} f \cos(\delta) \cos\left(\frac{\alpha}{2}\right) = \frac{2v}{\lambda} \cos(\delta) \cos\left(\frac{\alpha}{2}\right), \tag{2.14}$$

where f in Hz is the frequency at transmission, v in m/s is the target velocity vector, λ in m is the wavelength, c in m/s is the speed of light, δ in degrees is the aspect angle with respect to the bistatic bisector and α in degrees is the bistatic angle. As a matter of convention we say that a positive Doppler shift value represents an approaching target.

MTI Processing

Moving Target Indication processing addresses the goal of distinguishing the target from surrounding clutter. Clutter signals are echoes from objects in the environment, these echos are often as strong or stronger than that of the target, therefore simply increasing the transmit power of the radar will not improve the signal-to-clutter ratio (SCR). MTI processing is often spoken of in the context of ground clutter [15, 16], in the radar design stages the SCR can be improved by decreasing the antenna azimuth beamwidth, and in the case where the range cell extent is beam-limited, then also decreasing the elevation beamwidth. In other cases, where the range cell extent is pulse-limited or if volume clutter is a factor, improving the range resolution can also be beneficial. However, if the radar parameters are already fixed then Doppler processing is the most common means of improving the SCR.

Pulse-Doppler Processing

Pulse-Doppler processing addresses both the goal of separating the target from surrounding clutter, and determining the velocity [15]. This form of Doppler processing is particularly useful for this project and as such is considered in some detail. Before doing analysis on the spectrum of the data, a DFT is applied to each constant range bin in the fast-time/slow-time matrix. However, to

ensure that the slow time sequence can be properly reconstructed from the DFT spectrum, the spectrum size must be chosen to be greater than or equal to the original slow time samples. Furthermore, to avoid straddle loss we must further constrain the DFT samples to be strictly greater than the slow time samples [15].

Performing a DFT is essentially a form of coherent integration. Consider a moving target in a range bin within a coherent processing interval (CPI) of M pulses. If the target's velocity is constant with Doppler shift f_d [Hz] then the received signal can be modelled in slow time after quadrature demodulation as

$$y[m] = Ae^{j2\pi f_d mT}, \qquad m = 0, 1, .., M-1. \tag{2.15}$$

The signal has some amplitude A and is a function of the pulse repetition interval (PRI) here denoted by T. Then the DTFT of the above signal with respect to analogue frequency yields

$$Y(f) = A\frac{\sin\left(\pi(f-f_d)MT\right)}{\sin\left(\pi(f-f_d)T\right)}e^{-j\pi(M-1)(f-f_d)T}. \tag{2.16}$$

Note that the magnitude of the peak of this sinc function will be MA if we do not consider any windowing at this point. Specifically, if we now consider an input signal within white noise, we expect that the power of the noise will be spread uniformly in the frequency domain, while the energy of the signal to be concentrated with a peak MA , so in the frequency domain

$$SNR_{frequency} = \frac{\text{signal power}}{\text{noise power}} = \frac{|MA|^2}{M\sigma_n^2} = M(SNR_{time}). \tag{2.17}$$

Thus the factor M is commonly referred to as the coherent integration gain. A natural consequence of this is that the integration improves the SNR of the output, and a larger M is more favourable in that regard. In practice we make use of a DFT to compute the spectrum, if the signal is only in the presence of white noise, we can exactly achieve a gain of M given that white noise is distributed uniformly across all frequencies. However, we have neglected the effects of windowing, and in the case of NetRAD we also have to consider the

additional non-stationary sea clutter.

So in short we see that pulse-Doppler processing by way of DFT tries to separate target signals from other disturbances by considering their Doppler shift. The way this is achieved is to create a number of optimal filters, which are matched to the signal of a constant velocity target at different Doppler f_d [Hz]. In particular, if we consider the K-point DFT, then this set of filters, often also referred to as a Doppler filter bank, has discrete number of frequencies defined by

$$f_d = \frac{k}{KT} - \frac{k(PRF)}{K}, \qquad k \in \mathbb{Z}. \tag{2.18}$$

Now, for a particular Doppler f_d, we can create the optimum filter by defining the input vector $\mathbf{W}_p$, which has length M as shown in (2.15), and additionally padding it with $K - M$ zeros as

$$\mathbf{W}_p = \hat{A}\big[1 \;\; e^{j2\pi f_d T} \; ... \; e^{j2\pi f_d (M-1)T} \;\; 0 \; ... \; 0\big]^T. \tag{2.19}$$

Now if we only consider uncorrelated clutter, then we can say that the interference covariance matrix $\mathbf{S}$ is simply

$$\mathbf{S} = \sigma_n^2 \mathbf{I}_K. \tag{2.20}$$

Here $\mathbf{I}_K$ is the identity matrix of size K. Then we can compute the optimum filter coefficient vector $\mathbf{H}_{opt}$ as

$$\mathbf{H}_{opt} = \mathbf{S}^{-1}\mathbf{W}_p^*. \tag{2.21}$$

To compute the filter output we can apply this filter to the data, we must also define the slow time input vector, which is defined as

$$\mathbf{Y} = \big[y[m] \;\; y[m+1] \; ... \; y[M-1] \;\; 0 \; ... \; 0\big]^T. \tag{2.22}$$

And finally, the output of the filter is found by

$$z = \mathbf{H}_{opt}^T \mathbf{Y} = \frac{\hat{A}}{\sigma_n^2} \sum_{m=0}^{M-1} y[m] e^{-j2\pi f_d mT}. \tag{2.23}$$

Altogether, the DFT has effectively implemented all K matched filters at once, each tuned to a different discrete Doppler frequency. Naturally, these filters have been defined to be equally spaced. Of course, we can easily change the number of filters by changing the size of the DFT, and the filter shape can be altered by using a window function, such that the SNR of a target with Doppler f, will be maximised with that filter with the closest center frequency f_d.

2.2.3 Radio Interference Suppression

This section will highlight only one type of both approaches to suppressing interference presented in the paper by Lord and Inggs [1], namely spectral estimation with coherent subtraction and filter methods. These will be discussed in more detail in sections to follow.

As was mentioned before NetRAD operates at S-Band a band, which is also occupied by several other services. In particular the system has been found to be susceptible to interference from wireless local area network (WLAN) and line of sight (LOS) links [6]. Restrictive regulations do not allow for the power in these bands to be increased, since it would essentially degrade the signal quality of other services, so investigating unwanted signal suppression is warranted. Unfortunately, the nature of the interference is not spiky like that of target or common forms of clutter, but covers the entire range recording at any one time. To add to this the interference is often several magnitudes above the expected noise power level. This chapter draws on the research done on Radio Frequency Interference (RFI) Suppression by Lord and Inggs [1]. Unfortunately, the research done was focused on VHF-UHF Synthetic Aperture Radar, however, good insight on how to approach the problem can be gained. From this paper RFI suppression essentially involves the following steps:

1. Finding an appropriate model of the interfering signals
2. Estimating the parameters of the signal using measured data
3. Using the estimated parameters to suppress the interference in the data

Modelling the RFI The first step to suppressing the interference is to find a model for the RFI in the environment and in the worst case estimating it. There are different methods of achieving this, however, these require a recording of the particular interference environment. The most common outcome of these methods results in a model of the interference, which is a superposition of "tones", the wideband signal and white noise.

Parameter Estimation If a model is available, then a critical parameter that needs to be estimated is the modulation time of the RFI, since the effectiveness of the model strongly depends on how long the RFI remains coherent in the relevant range line. In the case of Wi-Fi this is not possible as the modulation time varies.

As mentioned before there are different approaches to suppressing the interference in the environment. Naturally, each method has its advantages and disadvantages, these are completely dependent how the interference is defined and what is considered to be desirable.

Spectral Estimation and Coherent Subtraction

If the model of the RFI is assumed to be a superposition of sinusoids then spectral estimation methods have been found to be effective [1]. These methods require the frequency, the phase and the amplitude of each interfering sinusoid to be estimated and then subtracted from the recorded signal to leave only the wanted part of the signal. Provided these parameters do not vary too much within one pulse length, good suppression with low signal distortion can be achieved. The work presented in the thesis by Lord and Inggs [1] does not implement any spectral estimation methods, they are outlined here.

Sinusoidal Parameter Estimation: MLE The first approach in this class of suppression, which can be used to estimate these parameters needed for subtraction, is the maximum likelihood estimate (MLE) approach. However, given that the frequency, phase and amplitude for more than one sinusoid needs

to be estimated, iterative techniques with some initial values should be used. These initial guesses can usually be found from the available data by applying an FFT. The advantage of using the MLE approach is that we can find the MLE for a given data set numerically. This is done by finding the maximum value of a known function, that is the likelihood function [17]. If the parameter estimate is small enough and within a known interval, then we are guaranteed to find the MLE for that set of data. If the parameter of interest is not limited to a known finite interval, then finding the MLE using a grid search is no longer computationally feasible. Thus, the need for iterative maximization methods. Algorithms that achieve this rely on initial estimates, and are generally good at finding the MLE given the initial estimate is close to the true maximum. The problem with these iterative methods, is that we often do not have prior knowledge of whether convergence can be obtained, and even if it is, whether this is the true MLE. So the distinguishing factor of this estimation problem is that the function to be maximized is not known a priori. The function to be maximized is thus a random function.

The interference data is assumed to be of the form:

$$x[n] = A\cos(2\pi f_0 n + \phi) + w[n] \qquad n = 0, 1, ..., N-1. \tag{2.24}$$

We must have $A > 0$ and $0 < f < 1/2$, then if we had a way of finding the true MLE, the idea is to estimate the parameters of one of the "tones", subtract it and repeat iteratively, to remove most of the interference.

Since there are multiple unknown parameters we can make use of Fisher information matrix [17]. Define θ as a vector of the parameters we are trying to estimate, such that:

$$\theta = [A \ f_0 \ \phi]^T. \tag{2.25}$$

Then, the elements of the Fisher Information Matrix are given by:

$$[\mathbf{I}(\theta)]_{\mathbf{i},\mathbf{j}} = \frac{1}{\sigma^2}\sum_{n=0}^{N-1}\frac{\delta \mathbf{s}[\mathbf{n};\theta]}{\delta\theta_{\mathbf{i}}}\frac{\delta \mathbf{s}[\mathbf{n};\theta]}{\delta\theta_{\mathbf{j}}}. \tag{2.26}$$

Where $\mathbf{s}[\mathbf{n};\theta]$ is the signal of interest in this case $\mathbf{s}[\mathbf{n};\theta] = A\cos(2\pi f_0 n + \phi)$.

Sinusoidal Parameter Estimation: AR The word "autoregressive" in signal processing and statistics, refers to models in which the signal depends on itself in a linear way. Autoregressive parameter estimation is quite popular and most often used in speech and econometric signal processing, but could prove useful in this case as well. The nth sample of the estimate $\hat{x}[n]$ of the measured data can be found using the autoregressive (AR) process, such that:

$$\hat{x}[n] = -\sum_{k=1}^{p} a[k]x[n-k] + u[n], \tag{2.27}$$

where $p/2$ is the number of sinusoids to be estimated, $a[k] = a[1], a[2], ...a[p]$ is the autoregressive parameter that needs to be derived from the measured data for each pulse, and $u[k]$ is the signal plus noise.

A third parameter estimation method investigated is a parametric maximum likelihood (PML) method. However this method is better suited to SAR images and is therefore not considered here.

Filters

It is common to examine the spectrum of the contaminated signal, identify the interference spikes, which are usually several decibels larger than the signal of interest, and removing these interfering spikes with the use of a notch filter [1]. However, this approach is only effective if the interference is narrow band relative to the input signal, and the use of a notch filter often has unwanted and adverse effects on a radar system. Thus, adaptive filters become attractive as the can provide interference suppression and overcome the undesirable distortion introduced by notch filters.

LMS Adaptive Filter Adaptive filters are commonly used in interference suppression due to the favourable property that little or no prior knowledge of the signal or interference characteristics is needed and that the parameters of the filter can change automatically. There is, however, the underlying assumption that the interference is sinusoidal. The model of an adaptive filter is shown in the Figure 2.3.

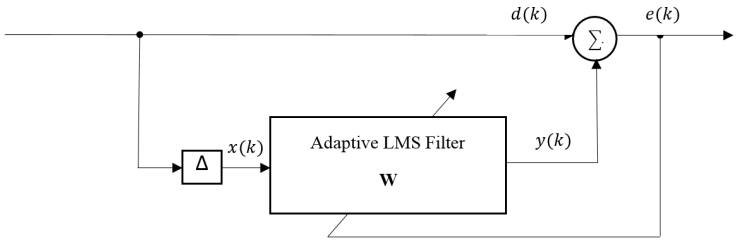


Figure 2.3: Block diagram of an adaptive LMS filter taken from Lord and Inggs [1].

This set up of the adaptive filter is used for interference cancellation makes use of an input signal here shown as $d(k)$, an interference reference signal input here represented by $x(k)$, which is simply a delayed version of the input $d(k)$. Then the filter sums and scales the inputs to produce an adaptive output here shown as $e(k)$. Note that the error is defined as $y(k) - d(k)$ and in this case is actually the "cleaned" desired signal. In other words, the error converges to the wanted input data signal, and not to zero.

So the n-element input signal vector D_k and the filter weight vector W are defined at time k as follows:

$$D_k = \{d(k), d(k-1), ..., d(k-n+1)\}, \tag{2.28}$$

$$W_k = \{w_0(k), w_1(k), ..., w_{n-1}(k)\}. \tag{2.29}$$

The reference signal, which in this case is just a delayed version of the input signal D_k, is denoted with the vector X_k, and is defined as,

$$X_k = D_{k-delay}. \tag{2.30}$$

The output of the filter is an estimate of the interference and is denoted by the scalar y_k and is defined as,

$$y(k) = W_k \cdot X_k^T. \tag{2.31}$$

As mentioned before the error $e(k)$ forms the desired signal and is defined as the difference of the contaminated input signal and the estimate of the interference as,

$$e(k) = d(k) - y(k) = d(k) - W_k \cdot X_k^T. \tag{2.32}$$

The algorithm then minimises the mean-square error by recursively altering the weight vector W_k according to the following relationship,

$$W_{k+1} = W_k + 2\mu e(k) X_k^*. \tag{2.33}$$

Where X_k^* is the complex conjugate of X_k and μ is the step size also known as the convergence factor or the adaptation coefficient, which controls the stability and the rate of adaptation. To find the upper bound of the step size μ we must calculate the maximum eigenvalue of the estimated autocorrelation matrix R. The article by Feuer and Weinstein [18] derives the result on the bound on μ quoted below

$$0 < \mu < \frac{1}{\lambda_{max}}, \tag{2.34}$$

where,

$$\lambda_{max} < N_{taps} \times \sigma^2. \tag{2.35}$$

σ^2 is the power in the signal and λ_{max} is the largest eigenvalue of the autocorrelation matrix. A larger value of μ increases the rate of convergence, but leads to a larger final misjudgement of the mean-squared error from the minimum mean-squared error that is produced by the optimum filter.

2.3 General-Purpose CPU Computing

The radar digital signal processing techniques discussed in the sections above depend largely on performing some operation on received signal data for all signals in a large dataset. To a large extent the processing on individual signals can be done independent of each other. Applications which exhibit such parallelism on a data level define a set of applications known as throughput computing applications [19]. The focus of such applications is to store and process massive amounts of data. Two main types of computing platforms exist suitable for this class of application, namely general-purpose central processing unit (CPU) and graphics processing unit (GPU). These two architectures are designed with different philosophies in mind. General-purpose CPUs have few cores relative to GPUs and are optimised to handle a wide variety of tasks with the fastest response time per task. The number of cores on a CPU is constrained by the price of complexity and power consumption [19]. GPUs are designed to handle a vast amount of smaller processing elements, and are best suited for graphics applications such as rendering, where the pixels exhibit a large degree of data parallelism. More recently in literature is the emergence of general-purpose GPU (GPGPU) [20]. The processing capability of GPU can be exploited in applications outside of the graphics domain provided the processing complexity per core is within bounds and the memory required per core can be held in the limited memory space. Ultimately both multi-core CPU and GPGPU implementations have proven to be successful in radar signal processing [20, 21, 22], although it remains to be seen whether the performance is comparable.

2.4 Julia Language for Radar Signal Processing

This project proposes the use of the Julia Programming Language (referred to as Julia for the remainder of the project) to develop signal processing infrastructure for NetRAD and NeXtRAD. In this section we will look at the motivation for using Julia and highlight some of challenges that are foreseen. Note that Julia

is a new programming language, first released to the public in February 2012 [23] and due to this there are currently no scholarly articles that make use of Julia in a radar signal processing context.

Julia is an open-source, high-level, high performance language targeted at scientific computing created by developers at MIT [24]. Even though Julia is still in beta form (stable release $v0.4.3$ when this was written) it already has a solid type system, which maps well to the primitive machine code, and has a working fully automatic type inference system, which lets the compiler deal with type declarations when the user does not want to do it [25]. Julia's development is fast growing with many of the popular libraries for Python, C and R already made available, although not all are complete.

When optimised, Julia's low level virtual machine based just-in-time compiler boasts performances comparable to C [24]. Julia is designed for parallelism and cloud computing, it provides a number of tools for distributed computation giving the user freedom to choose an implementation for their application as opposed to the traditional more rigid set of rules. Julia strongly resembles MATLAB in syntax, which makes quick prototyping possible, though the low-level implementation is vastly different. Julia lets the user define alternative implementations of the same function.

Some shortcomings of Julia are its libraries, even though much progress has been made to create some of the more popular libraries, it is still lacking considerably. Being a young language, Julia does not yet have a good development environment. Although there is an active online community, it is still difficult to find support on forums. Altogether the challenge will be to gain an understanding of how Julia works on a low-level to take advantage of its speed and power, and create libraries where none exist.

Chapter 3

NetRAD/NeXtRAD Sea Clutter Database

The data from the NetRAD campaign was collected from 2009 to 2011 in the United Kingdom and in South Africa [4]. An overview of the data collection trials and main objectives are summarised in Table 3.1 below.

Table 3.1: Table of NetRAD trials with summary of objective.

Date	Site	Size	Objective
11/2009	Peacehaven	132 GB	Sea clutter and range calibration
02/2010	Peacehaven	61 GB	Sea clutter and range calibration
08/2010	Peacehaven	198 GB	Sea clutter and range calibration
10/2010	Cape Peninsula	224 GB	Sea clutter and small targets
04/2011	Simon's Town and Misty Cliffs	165 GB	Small targets and VV bistatic sea clutter
06/2011	Misty Cliffs	269 GB	VV and VH bistatic sea clutter

During an experiment, the parameters generated by the multistatic system were either stored in a master file or on separate files for each monostatic node. In addition to this, extra video, audio, GPS time, weather and sea state information were also recorded for each experiment. The data has recently been reorganised into a standardised database in an HDF5 file format by researchers at University

of Cape Town [5]. HDF5 was created particularly for high performance, data intensive computing environments. The HDF5 file format was chosen for its well-defined data structures and data object management capabilities [4].

3.1 HDF5 Data Model and File Format

An HDF5 file is a container for HDF5 objects. Groups and dataset objects within a file are known as primary objects. A primary object has an absolute address stored within a file that is associated with it. A dataset contains an array of data elements along with supporting metadata. Datatype and dataspace objects form part of the data object definition, the datatype describes the elements within a data object array and the dataspace describes the dimension and rank of the data object array. To avoid redundancy with complex datatypes, they are treated as primary objects as the complete description of the datatype can be stored in the file and referred to when necessary [26]. Figure 3.1 shows the HDF5 object relationship hierarchy, and how they are each individually accessible.

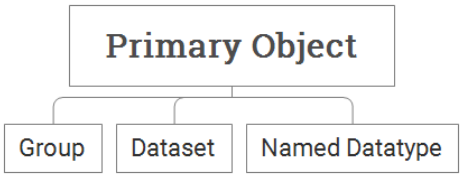


Figure 3.1: HDF5 Primary object hierarchy.

There are seven different classes of HDF5 objects, each of which are subclasses of the HDF5 object. The seven object classes are shown in Figure 3.2.

Groups and links create a structure and organisation among objects within a file in a *graph* with a single designated entry node, which is known as the root group. A file contains only one root group.

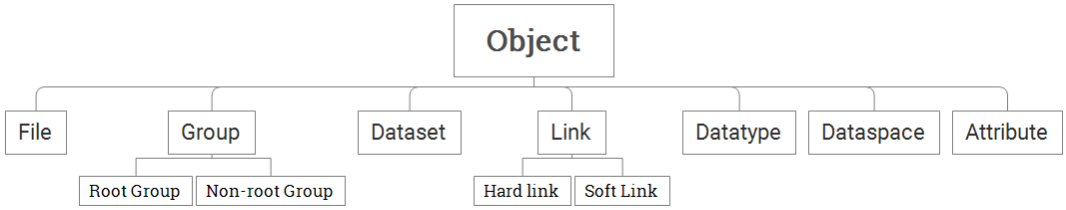


Figure 3.2: Seven HDF5 object types.

Links are a mechanism for finding members of groups. There are two types of link objects namely, a *hard link* and a *soft link*. Links represent an edge in a file graph structure, i.e. it connects a group with objects. A link's *target* is thus the object which is connected to a group through a link. A link has a (name, value) pair associated with it. The name is used as an identifier and the value is used to locate the link's target.

An *attribute* attaches content metadata to an object. In the NeXtRAD database the extra information such as sea state, radar parameters etc. are stored as attributes of a group within the file. Content metadata is used to describe the nature and/or the usage of the primary object it is associated with. This does not form part of the HDF5 object definition and can only be interpreted by the application. To access an attribute, they can be identified by a name or index value. Using the index value allows the user to iterate through all the attributes associated with a given object.

3.1.1 Julia HDF5 Interface

For simple types such as scalars, strings and arrays, HDF5 provides sufficient metadata to be able to interpret the items and are compatible across different computing architectures. JLD (*.jld) and MATIO (*.mat) modules automatically contain the necessary attributes to preserve Julia objects.

The Julia HDF5 Guide [27] provides a good overview of both low-level and high-level commands for file, object and data manipulation. The website

also provides some basic instructions on memory mapping, which is particularly useful for frequently used smaller data subsets. Direct memory mapping can be used to the bypass the HDF5 routines under particular conditions.

Note that HDF5 uses row-major order as opposed to Julia, which uses column-major order, as a consequence the dimensions of the arrays are inverted to that when using tools such as h5dump.

3.1.2 Data Acquisition Software

Keeping the overview shown in Figure 1.1 in mind, the objective in this section is to develop code which the end user can use to load in raw radar data from the compressed HDF5 file stored in the NeXtRAD database. Table 3.2 describes the available script for loading, navigating through and acquiring the radar raw data plus radar parameters.

Table 3.2: Table of available Julia scripts for HDF5 NetRAD data acquisition.

Script name	`NR_HDF5.jl`
Dependencies	None
Function	Opens HDF5 file, navigates to appropriate dataset and saves data to Julia matrix.

Algorithm Description This script demonstrates how to access and store datasets from the HDF5 files in the NetRAD database using the Julia HDF5 package. It is important to note that each HDF5 (*.h5) file contains the data for a particular experiment. The script starts by prompting the user for a HDF5 file (*.h5) name. The specified file is then opened in a "read only" mode. An overview of the structure of the file is then displayed on the screen. An example of a typical is shown below:

```
HDF5.HDF5File len 2
   Data:  HDF5.HDF5Group len 1
```

```
     SamplePerPulse_x_PulseNo:  HDF5Dataset (2048,200):  [0x1aeb,
0x1ee7,0x1e7b,0x1ea3,0x1e93
   Details:  HDF5.HDF5Group len 1
     Temp:  HDF5Dataset () :
```

The first line in the example above shows that it is an HDF5 file. `Data` and `Details` are group objects. `SamplePerPulse_x_PulseNo` is a dataset of size $(2048, 200)$, which belongs to `Data`. `Temp` is a dataset, which belongs to `Details`. The root group of the file is "/" and is stored in a variable `rootGroup` for convenience. The user can use the displayed information to retrieve the necessary dataset. This particular script reads in the dataset `SamplePerPulse_x_PulseNo` into a Julia variable called `data`, which can then be manipulated as a standard Julia array. The script also allows the user to display all the radar parameters and other additional information available for that particular experiment, which are stored as attributes in the `Temp` dataset.

3.2 Chapter Summary

The NetRAD data is stored in a standardised compressed file within a database. The HDF5 file format structure allows the NeXtRAD raw data to be stored as a dataset within an HDF5 group. Other information relevant to a data collection experiment is stored as attributes of a dataset labelled as "Temp". HDF5.jl is a Julia script which can be used to access the NeXtRAD data stored in these compressed HDF5 files. See Appendix D for some extra useful HDF5 commands.

Chapter 4

Radar Data Preprocessing Techniques

After the acquisition of the radar data together with radar parameters, the objective in this section is to create the tools necessary to provide "quick-looks" for sea clutter analysis. This chapter outlines the serial processing methodologies used for the preparation of the NetRAD data. Popular processing techniques such as pulse compression, Doppler processing and RFI suppression are developed in the Julia environment.

4.1 Design Considerations

Since the database for NetRAD and NeXtRAD was still under construction at the time of this paper, the datasets are assumed to be available in the form of stored bin/txt/csv files on the local disk. Juno, which is Julia's integrated development environment, was not used due to stability issues. This project is particularly concerned with the aid of research using NetRAD data and as a result, the scripts are structured as functions, all of which, make assumptions specific to the application of radar signal processing for NetRAD datasets. Julia's libraries are continuously being updated and restructured, thus this paper does

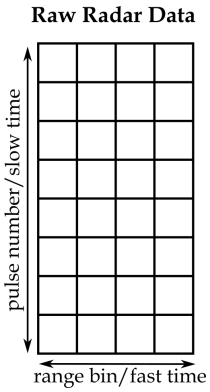


Figure 4.1: This is the shape of the matrix assumed in the pulse compression algorithms.

not try to find the most optimised way to implement the algorithms at this stage. The purpose of these scripts are therefore to demonstrate that Julia can be used for radar signal processing, and to provide a convenient means for preparing NetRAD data for further analysis.

4.2 Pulse Compression

This section deals with enhancing the SNR of NetRAD radar data using a serial pulse compression algorithm. The number of pulses of a particular NetRAD dataset is dependant on the measurement experiment. The largest dataset contains 130 000 pulses with 2 048 samples per pulse. The structure of the fast-time/slow-time raw radar data matrix referred to in following sections is shown in Figure 4.1.

It is important to read the raw data from the file into the correct shape. Julia uses a column-major rule.

4.2.1 Serial Pulse Compression Software

Table 4.1 gives a summary of the available scripts for serial pulse compression, it shows their dependencies, respective parameters and function.

Table 4.1: Table of available Julia scripts for serial pulse compression.

Script name	NR_PulseCompressSC
Dependencies	NR_ReadRefSig, NR_ReadBinFile
Function	Applies serial pulse compression routine on raw NetRAD radar data with input reference signal as the matched filter.
Script name	NR_PulseCompressionMMap
Dependencies	NR_ReadRefSig, NR_ReadBinFile
Function	Applies serial pulse compression routine on raw NetRAD radar data with input reference signal as the matched filter. Raw data is read from a memory mapped location on the disk rather than from RAM.

Algorithm Description The aim of NR_PulseCompress is to perform pulse compression on a NetRAD dataset. The dataset is assumed to be in a binary (*.bin) file format and is read into a (pulse number $\times$ range bin) matrix. The raw data from the binary file contains an offset, which is removed. The reference signal, which serves as the matched filter is read in from a text (*.txt) file. The reference signal is normalised, a Hanning window is applied, it is zero-padded and finally fast Fourier transformed. Note that Julia uses the Fastest Fourier Transform in the West (FFTW) as its default FFT routine [28]. The FFTW library has an optional function `planfft`() which determines the optimal implementation of the FFT for the given data.

Then if the Boolean parameter `analytic_` is set to false:

The conjugated reference signal `r` is multiplied with the FFT of each line of the data matrix sequentially in a loop `Npulses_` times. The output is stored to another matrix. Each line of the processed matrix is then inverse fast Fourier

transformed to form the final output matrix `dOut`.

Otherwise, if the Boolean `analytic_` is true:

The analytic signal `dPositiveHalf` is formed. The conjugated reference signal `r` is multiplied with the FFT of each line of the analytic data matrix sequentially in a loop `Npulses_` times. Each line of the processed matrix is then inverse fast Fourier transformed to form the final output `dOut`.

The algorithm is also presented in pseudocode in Algorithm 1.

Algorithm 1 NR_PulseCompresSC: NetRAD Pulse Compression

```
1: function SCPC(nPulses_, rawDataFile_, refSigFile_, analytic_)
2:     d ← raw data from (rawDataFile_)                ▷ see NR_ReadBinFile
3:     r ← reference signal from (refSigFile_)         ▷ see NR_ReadRefSig
4:     P ← fft plan on d                  ▷ find best FFT implementation for d
       ▷ apply planned FFT on each pulse
5:     for i ← 1, nPulses_ do
6:         dMatched ← r. ∗ (P ∗ d)
7:     end for
8:     P ← ifft plan on dMatched          ▷ find best IFFT implementation for
   dMatched
9:     if analytic_ == false then
       ▷ apply planned IFFT
10:        for i ← 1, nPulses_ do
11:            dIFFT ← r. ∗ (P ∗ d)
12:        end for
13:        return dIFFT
14:    else
15:        dPositiveHalf ← analytic signal
16:        dOut ← ifft of dPositiveHalf
17:        return dOut
18:    end if
19: end function
```

4.2.2 Results and Discussion

The pulse compression routine requires large data arrays to be stored in RAM during the routine. Julia gives the user little control over memory management. In particular, Julia does not have a method to free memory in RAM. As a result, the performance of the system may suffer if all the available RAM is used. Table 4.2 outlines the RAM required to perform pulse compression on 130 000 pulses of NetRAD data. In the event that limited RAM is available, some methods of

Table 4.2: Table outlining variable memory usage in serial pulse compression routine.

Variable Name	**Variable Size**	**Variable Type**	**Required Memory**
`d`	$(130\,000, 2\,048)$	$Array\{Float32, 2\}$	1 015.5 MB
`r`	$(1, 2\,048)$	$Array\{Complex\{Float32\}, 2\}$	0.016 MB
`dPositiveHalf`	$(130\,000, 1\,024)$	$Array\{Complex\{Float32\}, 2\}$	1 015.5 MB
`dMatched`	$(130\,000, 2\,048)$	$Array\{Complex\{Float32\}, 2\}$	2 031 MB
`dOut`	$(130\,000, 2\,048)$	$Array\{Complex\{Float32\}, 2\}$	2 031 MB
TOTAL			6 093.02 MB

dealing with this are considered. The most straightforward solution to a memory issue is to reduce the number of pulses considered in an experiment. However, some other methods include garbage collection and memory mapping.

Garbage Collection The only available method of clearing memory in Julia v0.4.2 is to force a so called garbage collection. The method is to set the variable that needs to be cleared to zero and then to use the `gc()` command to force the garbage collector to clear the memory. However, forcing a garbage collection can add significant amounts of overhead in program computation time.

Memory Mapping In Julia, memory mapping is a method where non-primary is memory treated as if it were located in the RAM. An immediate repercussion of this is that the access time of disk memory will not be as fast as that of RAM. In other words, the trade-off is computation time. The script NR_PulseCompressionMmap makes use of memory mapping and is available if

the full dataset is required and processor memory is limited.

Consider the trend of computation time as a function of dataset size. Naturally, we should expect the program to take longer to run when considering more data. This is evident in Figure 4.2 below.

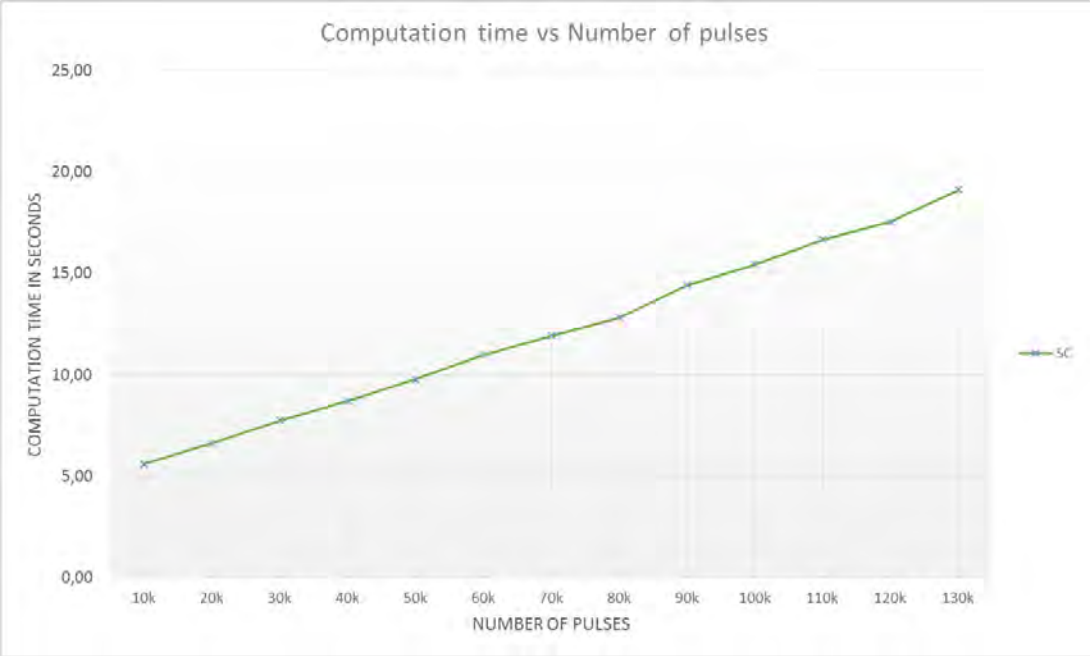
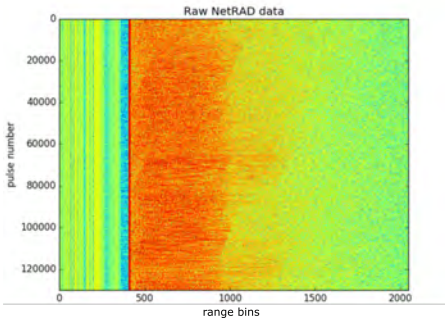


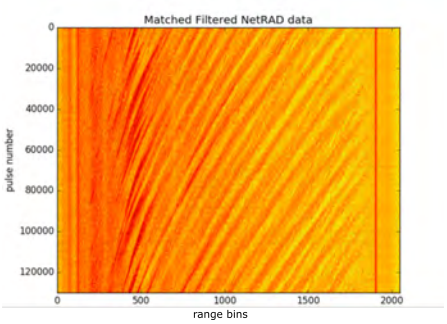
Figure 4.2: Plot of serial pulse compression computation time as a function of number of pulses.

As expected the computation time increases linearly with the number of pulses. It took roughly 19 seconds to perform pulse compression on 130 000 pulses and 2 048 samples per pulse.

Pulse Compressed Quick-Look: RTI plot The pulse compressed data can be visualised with what is known as a range-time intensity plot (RTI plot). It is a 2-dimensional contour plot, which has the slow time on the y-axis, the fast time on the x-axis and the amplitude of each pulse as a range of colour. Figure 4.3 shows a typical RTI plot, which serves as a quick-look for raw and match filtered data respectively.



(a) An example of an RTI plot of the unfiltered NetRAD dataset.



(b) An example of an RTI plot of the pulse compressed NetRAD dataset.

Figure 4.3: An example of an RTI plot before and after pulse compression. The scene is of the ocean, the high intensity curves in (b) indicate waves moving toward the radar over time.

4.3 Doppler Processing

It is common practice to apply Doppler processing after pulse compression as the output SNR is significantly increased. Unlike ground clutter, sea clutter is not stationary, so MTI processing will not provide any meaningful knowledge to an observer. Pulse-Doppler processing provides information on the speed of potential targets in the data. Thus it is important that the SNR be as high as

possible. This section describes the continued preparation of the NetRAD radar data, using pulse-Doppler processing.

4.3.1 Serial Pulse-Doppler Processing Software

Table 4.3 gives a summary of the available script for serial Doppler processing. Refer to the accompanying manual for the scripts outlined.

Table 4.3: Table of available Julia script for Doppler processing.

Script name	NR_DopplerProcessingSC
Dependancies	NR_PulseCompressSC
Function	Applies serial pulse-Doppler processing and pulse compression on raw NetRAD data with variable DFT size.

Algorithm Description NR_DopplerProcessingSC defines a function which requires the number of pulses, the DFT size, the raw radar data and the reference signal. The function then calls the serial pulse compression routine NR_PulseCompressSC and saves the pulse compressed data. The data is then transposed and a DFT of size `ds_` is applied to all elements of each constant range bin. The function returns `s`, a matrix with Doppler processed data.

The algorithm is also presented in pseudocode in Algorithm 2.

4.3.2 Results and Discussion

Again we consider the computation time of the pulse-Doppler processing script for increasing number of pulses as shown in Figure 4.4.

Again as expected, similar to the case with pulse compression, processing more pulses results in longer computation times. In this case, the number of

Algorithm 2 NR_DopplerProcessSC: NetRAD Doppler Processing

```
1: function NR_DopplerFilter(np_, ds_, inputData_, inputR_)
2:     data ← pulse compressed inputData_          ▷ see NR_ReadBinFile
3:     define s                          ▷ complex array with Doppler filter size
4:     r ← reference signal                          ▷ see NR_PulseCompress
5:     a ← transpose data                ▷ find best FFT implementation for data
6:     P ← fftplan on s
       ▷ apply windowed planned FFT on each range line
7:     for n ← 1, size of range samples do
8:         for m ← 1 : ds_ : np_ do
9:             s ← fftshift(P ∗ (a[FFT window]))
10:        end for
11:    end for
12:    return s
13: end function
```

pulses is a function of the DFT size (in this case 256). It took roughly 53 seconds to process 122 880 pulses.

Doppler Processed Quick-Look: RTI plot When a target of interest is observed in a particular pulse of the pulse compressed RTI plot, a range-Doppler plot can then be processed to give information on the Doppler in that range bin. If the target is not visible in the pulse compressed RTI plot, series of range-Doppler RTI plots can be made for all successive pulses. On the x-axis is the DFT bins, which can be converted to a range of Doppler frequencies, i.e. the x-axis goes between $[-f_d,\ f_d]$ with the center of the x-axis corresponding to the 0 Hz Doppler bin. On the y-axis are range bins. The plot gives an indication of the amplitude of the Doppler of targets at a particular range over a short instance in time. Figure 4.5 is an example of the range-Doppler RTI plot for the first 256 pulses.

Figure 4.4: Graph of computation time of the serial pulse-Doppler processing script as a function of the number of pulses processed.

4.4 Radio Frequency Interference

Besides sea clutter analysis and small target detection, the NetRAD system was also used in other experiments involving drone detection and tracking [29], and target classification [30]. The aim of one experiment was to see whether NetRAD could detect a small drone with varied radar cross sections. The geometry of the set-up consisted of the three nodes spaced 40 m apart and the target at a distance of 65 m from node 1 as illustrated in Figure 4.6.

Initially, four test cases are examined:

- *case* 1 - no drone present (empty background)
- *case* 2 - no drone present, cardboard tube supports
- *case* 3 - drone stationed on cardboard tube supports
- *case* 4 - drone wrapped in tin foil stationed on cardboard tube supports

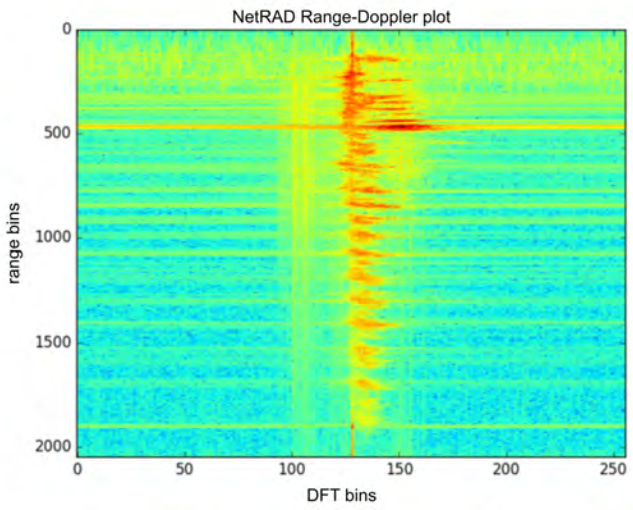


Figure 4.5: Plot showing the range and amplitude of Doppler frequencies at different range bins. Here there are several targets all with positive Doppler, corresponding to the waves in Figure 4.3b

In each case the range profile data is collected and compared. Once the initial tests proved that the drone could be detected without the need for tin foil to increase the RCS, the target drone was then successively moved further away from node 1 to find the maximum range of detection. Further tests included switching on the drone and removing the cardboard support structure. RTI plots of the drone on monostatic node 3 were recorded. The drone was remote controlled using Wi-Fi. During experiments, significant interference due to the remote controlled signals as well as other Wi-Fi communications were evident in the recorded data. This is motivation for interference suppression. This section highlights the results obtained from an adaptive Least Mean Squared (LMS) filter designed to suppress the RFI present in the NetRAD datasets from this experiment. To quantify the effectiveness of the filter, an Ordered-Statistic Constant False Alarm Rate (OS-CFAR) detector is applied to selected pulse data and the precision and recall for a fixed probability of false alarm (P_{FA}) before and after the application of the filter is compared.

The Figure 4.7 below shows the output of the matched filter before filtering, which is contaminated with Wi-Fi interference signals.

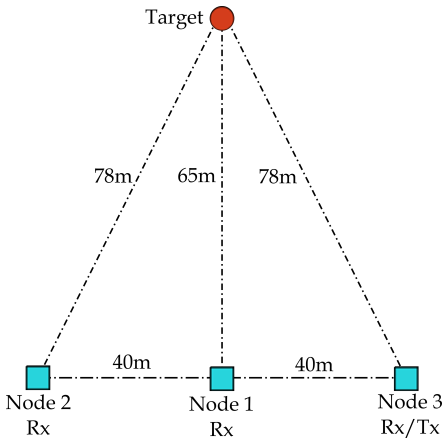


Figure 4.6: Illustration showing the geometrical set-up of the NetRAD nodes with respect to the target of interest.

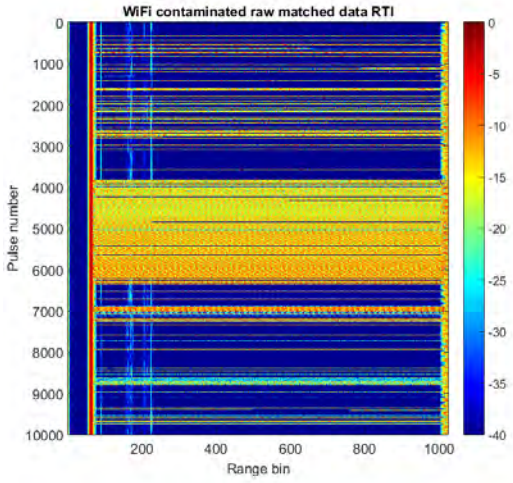


Figure 4.7: Figure showing Wi-Fi contaminated NetRAD data. The Wi-Fi appears as the lighter horizontal lines which effectively masks the underlying target returns.

Ideally the filter should suppress the stronger Wi-Fi signals without degrading the target signals hidden beneath. Note that the target signal amplitudes are significantly weaker than the interference signals. As was mentioned in the review of the work by Lord and Inggs [1] in Section 2.2.3 the decision is to use an LMS filter as it requires no *a priori* knowledge of the interference. However, unlike in the case seen in the reviewed work, applying the filter to samples

from a particular pulse was found to be ineffective. In this case the interference signal saturates the receiver for all fast-time samples in a pulse. The result is that the filter cannot discriminate between the estimated interference and the target signals. Consider pulses 5 150 to 5 200 shown in Figure 4.8. Notice that there are sections of uncontaminated signal between contaminated ones. In light of this, the filter is applied in the pulse number dimension, where it has the opportunity to see both clutter and signal.

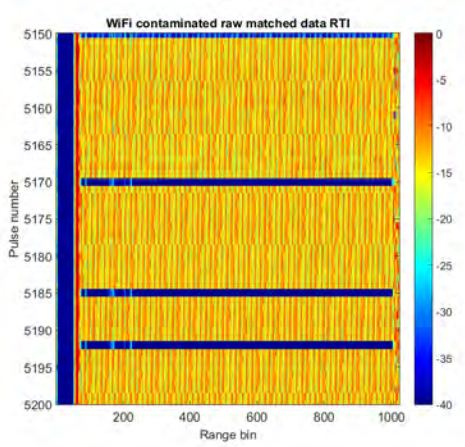


Figure 4.8: Figure showing that there are patches of clean signal within contaminated pulses.

4.4.1 Interference Suppression

Table 4.4 gives a summary of the available script for interference suppression. Refer to the accompanying manual for the scripts.

Table 4.4: Table of available Julia scripts for interference suppression

Script name	NR_InterferenceSuppression
Dependencies	None
Function	Applies serial adaptive LMS filter to match filtered NetRAD data.

Algorithm Description The algorithm reads in the matched filtered data and processes each constant range bin line iteratively. Within one range line a section of size `n` is considered as the input signal `X`. The error signal is computed to be matched data signal at time k minus the product of the input signal `X` and the filter weights vector `W`. At each iteration, new filter weights are calculated and the error is recorded as the output. This is repeated for all the pulse number bins `p`.

The algorithm is presented in pseudocode in Algorithm 3.

Algorithm 3 NR_InterferenceSuppression: NetRAD Interference Suppression

```
1:  procedure NR_InterferenceSuppression
2:      open mat file        ▷ from Julia's MAT package to read in MATLAB files
3:      data ← data from file                                  ▷ read from the file
4:      close file
5:      choose p                                               ▷ number of pulses
6:      choose n                                       ▷ number of filter weights
7:      choose d                                               ▷ size of delay
8:      choose mu                                                ▷ step size
9:      define W                                                ▷ filter array
10:     for k ← 1,range samples do
11:         for j ← p − n do
12:             X ← window of data, size n
13:             find the error
14:             update the weights
15:             save the error signal
16:         end for
17:     end for
18: end procedure
```

The results that follow are obtained using MATLABs R2015a `phased.CFARDetector` function with the parameters listed below:

`Method`	`'OS'`
`Rank`	8
`NumCuardCells`	2
`NumbTrainingCells`	12
`ThresholdFactor`	`'Auto'`
`ProbabilityFalseAlarm`	0.001
`ThresholdOuputPort`	`true`

The detector returns an array [`result`, `thresholdOutput`]. The first element is a binary array, of which each element is a 1 if a detection is made based on the adaptive threshold, and 0 if no detection is made. The second element is the value of the threshold for each range bin of every pulse.

4.4.2 Results and Discussion

As mentioned in the literature review, it is necessary to ensure that the filter weights `W` converge to a steady-state estimate as the time variable k tends to infinity. This can be satisfied if the adaptation coefficient, μ is carefully chosen to be in the following range,

$$0 < \mu < \frac{1}{\lambda_{max}}. \tag{4.1}$$

The μ value, filter weights and delay were optimised for dataset 1 shown in Figure 4.7 and verified on two others. All parameters of the filter were kept the same for each experiment, and are listed below:

μ	1.8×10^{-6}
filter weights	60
delay	2

The effectiveness of the filter is evaluated visually and by calculating the precision and recall before and after applying the adaptive LMS filter. Refer to Figure 4.5 for a definition if the decision rule.

Table 4.5: Definition of Decision Rule for Detection

		Decision	
		predicted condition true	**predicted condition false**
Hypothesis	**target**	true positive	false negative
	no target	false positive	true negative

Here a *true positive* implies the target is correctly detected. A *false positive* is a false alarm (type I error), a *false negative* is missed detection (type II error) and a *true negative* is when a "no target" is correctly classified.

The precision can be interpreted as how often the detector makes target detection which is correct, that is, how often is a detection a true positive. Mathematically it is calculated as:

$$\text{Precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}. \tag{4.2}$$

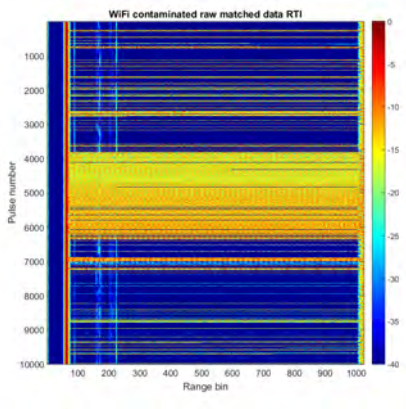
In this case the recall is a measure of the sensitivity of the detector, that is, how often it detects the presence of a target, given a target is present. We calculate this as:

$$\text{Recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}. \tag{4.3}$$

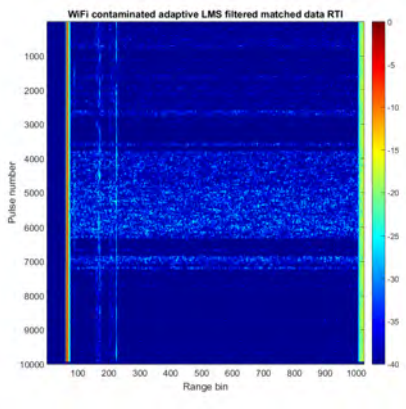
In each case we will classify a pulse to have *High Clutter* if the pulse exhibits a spiky profile for all range bins and *Low Clutter* if the pulse contains mostly signals from the targets. The results for the precision and recall for a High Clutter and Low Clutter pulse for each dataset is tabulated in Tables 4.6, 4.7 and 4.8 on page 50.

In Figure 4.9 dataset 1 is considered. Applying the filter to the dataset shown in Figure 4.9a yields the output shown in Figure 4.9b.

A visual improvement in the signal to clutter ratio is observed. The target signals are now visible behind the highly cluttered area around pulses 4 000 to 6 000. In this area there is roughly a 25 dB suppression of the Wi-Fi interference while the target signal has been suppressed by 15 dB. When the OS-CFAR is set to try



(a) Figure showing the heavily Wi-Fi contaminated dataset on which the adaptive LMS filter is designed.



(b) Figure of dataset shown in 4.9a after the suppression filter is applied.

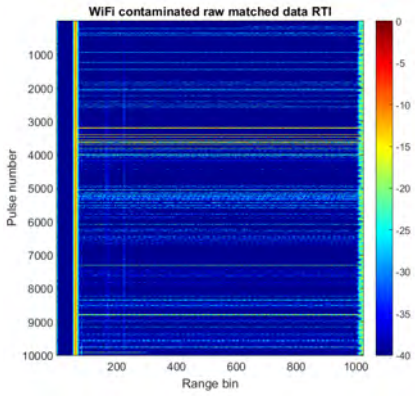
Figure 4.9: RFI dataset 1 (a) before and (b) after adaptive LMS filter interference suppression

and maintain a P_{FA} of 0.001, the detections and calculated precision and recall are summarised in the Table 4.6.

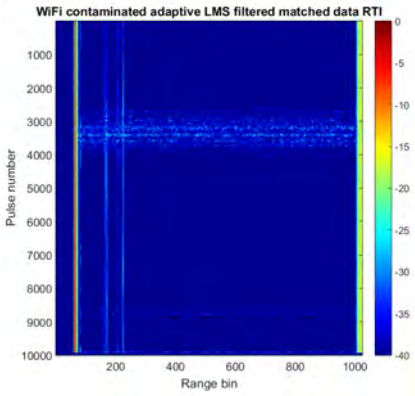
In a highly cluttered pulse the filter is able to achieve a precision on 100% by correctly detecting either the second or third targets, with a recall of 33% as it only detects one of three targets. In Low Clutter the detector is able achieve 100% precision and recall for this dataset.

The results for two other less contaminated datasets are shown in the Figures 4.10 and 4.11. Dataset 2 shown in Figure 4.10 yielded good overall interference suppression at roughly 20 dB while the target is only suppressed by roughly 8 dB, however, in this case the resulting interference signal strength is roughly the same as the target signals.

As expected, after filtering, in High Clutter the detector is still unable to distinguish between the signal and suppressed interference. However, the more sparsely cluttered areas are well suppressed. In Low Clutter the filter is able to achieve an improved precision, but is less sensitive to targets. Overall the filter is able reduce the number of High Clutter pulses.



(a) Figure showing the contamination level of verification dataset 2 before filtering.



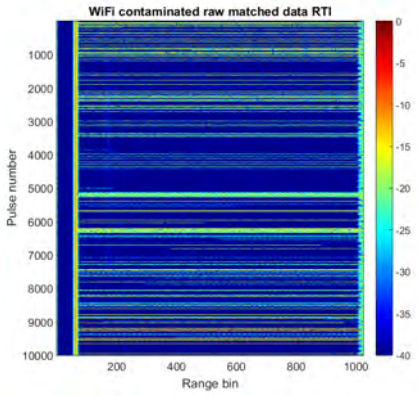
(b) Figure of dataset shown in 4.10a after suppression filter is applied.

Figure 4.10: RFI dataset 2 (a) before and (b) after adaptive LMS filter interference suppression

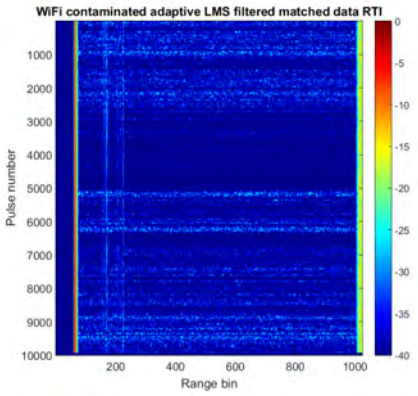
The third dataset shown in Figure 4.11 had interference spread uniformly across most pulses. In this dataset the drone target is not visible before filtering as can be seen in Figure 4.11a.

The filter yielded an interference suppression of roughly 8 dB, although again the filtered interference signal strength is similar to the target signals, and in addition, the interference still covers data across most pulses. The detection results are shown in Table 4.8.

Again, as expected, the detector is unable to detect the presence of a target in highly cluttered areas. Given that the first target is not visible in the matched filtered data before interference suppression, the maximum recall achievable is 67%, which was achieved for Low Clutter pulses before and after the application of the LMS filter.



(a) Figure showing the contamination level of verification dataset 3 before filtering.



(b) Figure of dataset 3 after suppression filter is applied.

Figure 4.11: RFI dataset 3 (a) before and (b) after adaptive LMS filter interference suppression

4.5 Chapter Summary

This chapter presents radar processing techniques that form part of the software toolbox for NetRAD and NeXtRAD development. Each section highlighted the available scripts for the particular tool, provided a description of the code and summarised the results of applying the tools to NetRAD data.

Section 4.2 described the serial pulse compression tool. It highlighted Julia's memory management shortcomings and proposed a memory mapping method if a user does not have access to more memory. The computation time of the script for an increasing number of pulses in the dataset was considered. NetRAD has a pulse repetition interval of 1 ms, i.e. it collects 130 000 pulses of data in 130 seconds. Given that the algorithm processed the entire dataset in under 20 seconds, the results show that given sufficient RAM the script could be modified to work in real-time.

Section 4.3 described the Doppler filter tool. The script implemented a pulse-Doppler filter rather than an MTI as NetRAD datasets were assumed to be of the sea. Sea clutter is not stationary and therefore pulse-Doppler processing

can give information on the location and Doppler of potential targets. Here again, the computation time for varying number of pulses was considered and found to be fast enough for a real-time application.

Given the unwanted consequence of operating in S-Band at 2.4 GHz, NetRAD has to deal with clutter in the form of Wi-Fi interference. The aim of Section 4.4 was to implement a post-processing interference suppression technique. An adaptive LMS filter was applied to NetRAD datasets from a drone detection experiment. The results show that using an OS-CFAR detector maintaining a $P_{FA} = 0.001$, the LMS filter is able to increase the precision in cases where the clutter is more concentrated in particular zones and not spread throughout the dataset. Even when the clutter is suppressed significantly, the detector was still unable to distinguish target from clutter in High Clutter pulses.

Table 4.6: Summary of OS-CFAR results for dataset 1 with 3 targets

Clutter	**Metric**	**Before filter**	**After filter**
High Clutter	Number of detections	0	1
	Number of true positives	0	1
	Number of false positives	0	0
	Precision	0	1
	Recall	0	0.33
Low Clutter	Number of detections	3	3
	Number of true positives	3	3
	Number of false positives	0	0
	Precision	1	1
	Recall	1	1

Table 4.7: Summary of OS-CFAR results for dataset 2 with 3 targets

Clutter	**Metric**	**Before filter**	**After filter**
High Clutter	Number of detections	2	0
	Number of true positives	0	0
	Number of false positives	2	0
	Precision	0	0
	Recall	0	0
Low Clutter	Number of detections	2	1
	Number of true positives	1	1
	Number of false positives	1	0
	Precision	0.5	1
	Recall	0.5	0.33

Table 4.8: Summary of OS-CFAR results for dataset 3 with 3 targets

Clutter	**Metric**	**Before filter**	**After filter**
High Clutter	Number of detections	0	0
	Number of true positives	0	0
	Number of false positives	0	0
	Precision	0	0
	Recall	0	0
Low Clutter	Number of detections	2	2
	Number of true positives	2	2
	Number of false positives	0	0
	Precision	1	1
	Recall	0.67	0.67

Chapter 5

Julia Parallel Computing

An algorithm can usually be broken into a section that can only be processed in serial (i.e. on one processor) and a section that can be split up and run on more than one processor at once. If the most computationally expensive part of the code can be split amongst several processors, then the overall computation time can be significantly improved. In the case where the most computationally expensive part cannot be split up, there is little gain in parallelising the algorithm. Amdahl's Law [31] gives a mathematical expression for the amount of speedup that can be achieved as:

$$\text{Speedup} = \frac{s+e}{s+e/p} = \frac{1}{s+e/p}. \tag{5.1}$$

Where, s is the average time to compute the serial part, e is the average time to compute the part that can be done in parallel and p is the number of cores.

Most computer systems today work with a local set of CPUs. A growing trend in Radar Signal Processing Systems is to use General Purpose Graphics Processing Units (GPGPUs) to perform high speed parallel algorithms [20, 32]. Julia parallel computing relies on the use of multiple CPUs [33] rather than GPUs, however, development to integrate this capability is in progress [34, 35]. Two main factors that determine the performance in parallel computing is the speed of the individual processors and the speed at which they can access their allocated

memory. For the best performance in a multi-core environment memory needs to be finely controlled.

Julia's multiprocessing environment is based on a unique message passing system which allows programs to be run on several processors in separate memory domains [33]. The implementation is unique as it is "one-sided", which gives the programmer control of only one process in a two process operation. The operations required to achieve this resemble high-level operations such as function calls and the use of macros rather than the conventional "message send" and "message receive" operations.

Julia's parallel programming is built on two primitives: remote references, which are objects accessible by any process to refer to its stored object on a particular process, and remote calls, which return a remote reference to its result.

Julia's base installation comes standard with built-in support for a local cluster and clusters across multiple machines [24].

Data Movement The major contributor to overhead in a parallel program comes from data movement and message sending. Reducing the number of messages sent and the amount of data moved between processes can significantly improve the runtime performance.

Distributed Arrays and Shared Arrays To perform pulse compression and Doppler processing on an array of raw radar data, we must introduce the concept of a distributed array. A distributed array is one that can be split such that specified processes have access to only a portion of the array. A shared array uses the systems shared memory to map the same array to many different processes. Thus, as the name suggests each processor has shared access to the entire array. Currently both these capabilities are under construction in the latest build of Julia.

Julia Processes In Julia each process has an identifier associated with it. The process that provides the interactive Julia prompt has ID 1, all other processes are called workers. If there is only one process, then process 1 is considered a worker.

5.1 Multi-core Pulse Compression Software

The rationale behind developing multi-core pulse compression is to take advantage of the fact that it is in essence a correlation operation of every range line with the matched filter. In serial, the algorithm does this correlation sequentially in a loop. The multi-core algorithm with p cores, attempts to call a serial correlation function on evenly distributed range line data. Thus, the idea is to cut the processing time of the correlation by p. This technique, however, has a draw-back. It is commonly acknowledged that the overhead associated with starting up parallel processes, as well as the limited memory pool and slow data movement between processes introduce significant overhead to computation time. It is therefore important to acknowledge that these practical constraints can result in the parallel implementation effectively performing slower than the serial implementation.

Consider the serial pulse compression algorithm. The part of the code that is done in serial is roughly 1/3 of the total time required to run the program. Thus, using (5.1) the speedup we should expect for $p = 2$ is 1.54 and for $p = 4$ is 2.05.

Figures 5.1 and 5.2 illustrate the overall flow of the data for a serial and a multi-core pulse compression implementation respectively. This section describes the parallel implementation of the pulse compression algorithm and highlights some of the results obtained in terms of computation time, number of cores and number of pulses considered. Table 5.1 on page 55 shows the script available for multi-core pulse compression.

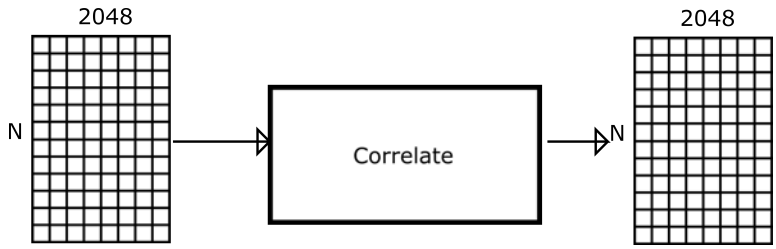


Figure 5.1: A figure illustrating the overall flow of data in the serial pulse compress algorithm described in NR_PulseCompressSC.jl.

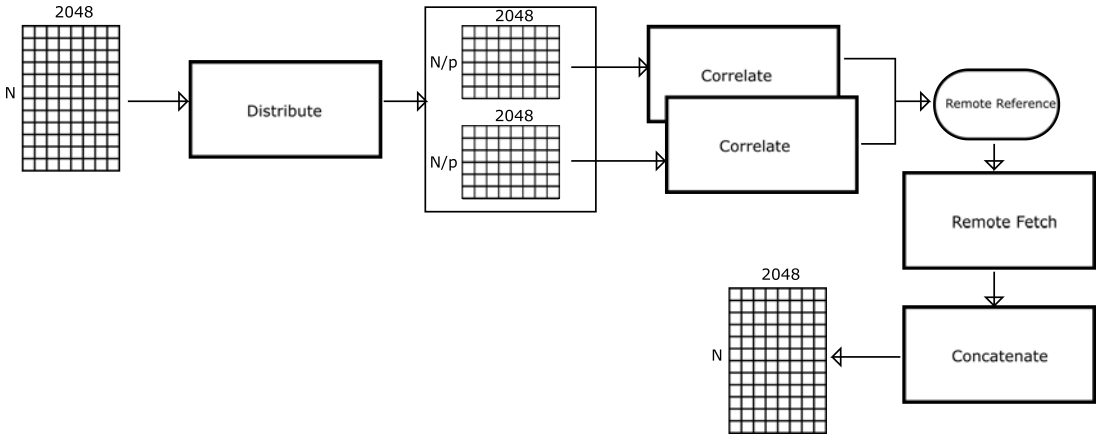


Figure 5.2: A figure illustrating the overall flow of data for a 2 core pulse compression algorithm described in NR_PulseCompressionMC.jl.

5.1.1 Multi-core Pulse Compression Code

Algorithm Description The script NR_PulseCompressionMC, also expressed in pseudocode in Algorithm 4 and Algorithm 5, aims to make use of an internal cluster of processes using Julia's multi-core platform. It starts by defining the function `NR_Distribute()`, which takes a local array and splits it into equal parts along its two dimensions defined by a tuple (row,col). Julia has an in-built distribute function, however, it does not give the user the ability to split the matrix into custom dimensions.

Next the serial function `NR_Correlate()` defines the pulse compression routine, which is to be run on p cores simultaneously. Note that `NR_Correlate()` is made available to all processes via the `@everywhere` macro, and takes in a Boolean `analytic_`, which optionally returns the analytic signal data matrix necessary if Doppler filtering is to follow the pulse compression.

The function `MCPC()` (Multi-core Pulse Compression) reads in the processed reference signal and radar data. Next the radar data must be distributed into `P` equal parts along the pulse number dimension. Once the data in memory is declared a distributed array, the data assigned to each worker is moved to its respective worker process. Next the `NR_Correlate()` function is invoked on each worker process and performs the correlation on the chunk of data associated to it. The `@spawnat` macro returns remote references, which specifies a location in memory, where the solution to `NR_Correlate()` is held.

Table 5.1: Table of available Julia scripts for Pulse Compression

Script name	NR_PulseCompressionMC
Dependancies	NR_ReadBinFile, NR_ReadRefSig
Function	Applies multi-core pulse compression, optionally creates the analytic signal necessary for Doppler processing. Returns either the analytic signal matrix or the matched filtered matrix.

Finally, the references are fetched using a parallel map (`pmap()`) and all the data is collected in an array of (`nPulses_`/`P`, 2 048) arrays.

Algorithm 4 NR_PulseCompresMC: NetRAD Pulse Compression Part 1

▷ this function must be made available to all processes using @everywhere
function NR_Correlate(np_, refSig_, rawData_; analytic_)
    $d \leftarrow$ read data from $rawDataFile_$ ▷ see NR_ReadBinFile
    $r \leftarrow$ reference signal from $refSigFile_$ ▷ see NR_ReadRefSig
    $pl \leftarrow$ fft plan on d ▷ find best FFT implementation for d
    for $i \leftarrow 1, np_$ **do** ▷ apply planned FFT
        $d \leftarrow r. * (pl * d)$
    end for
    $pl \leftarrow$ ifft plan on d ▷ find best IFFT implementation for data
    if $analytic_ ==$ true **then** ▷ create analytic signal
        create analytic signal
    end if ▷ assume the analytic signal is formed
    for $n \leftarrow 1, np_$ **do** ▷ apply planned IFFT
        $d \leftarrow r. * (pl * d)$
    end for
    return d
end function
function NR_Distribute(A, mydist, myprocs) ▷ A modified version of Julia's distribute function
    $owner \leftarrow$ current ID
    $rr \leftarrow$ remote reference
    store A on rr
    define $DArray$ on $myprocs$ number of processes into $mydist$ rows and cols
    for $DArray$ **do**
        for every processor fetch designated distributed part of $DArray$
    end for
    return d
end function

Algorithm 5 NR_PulseCompresMC: NetRAD Pulse Compression Part 2

27: **function** MCPC(nPulses_, rawDataFile_, refSigFile_, analytic_)
28: $P \leftarrow nprocs() - 1$ ▷ number of available processes
29: $PCchunk \leftarrow$ size of chunk
30: define $Ddata$ ▷ a distributed array
31: $dRaw \leftarrow$ read data from $rawDataFile_$
32: $rSig \leftarrow$ read reference signal from $(refSigFile_)$
33: $Ddata \leftarrow$ distribute $dRaw$ ▷ distribute the $dRaw$ into P number of chunks in pulse dim
34: **for** $w \leftarrow 1, P$ **do**
▷ assigns only one chunk to each process
35: $refs \leftarrow$ spawn the chunks from Ddata on all processes and perform NR_Correlate, return remote reference
36: **end for**
37: define $DdataArray$ ▷ array to hold the pulse compressed chunks of data
38: $DdataArray \leftarrow$ fetched remote references ▷ fetched in a parallel map
39: **return** $DdataArray$
40: **end function**

5.1.2 Results and Discussion

The details of the hardware are shown in Table E.1 and the results for multi-core pulse compression are discussed below.

To test whether the multi-core algorithm does in fact speed up computation and to determine at which point this occurs, the algorithms were run on an increasing number of pulses. In each case computation time was recorded. The results are shown in the figure below.

Figure 5.3 is a graph which shows the recorded computation time of the serial and multi-core algorithms as a function of pulses processed. The results show that unexpectedly the serial algorithm outperformed the multi-core algorithm in both the 2 core and 4 core test for almost any number of pulses considered. The graph also shows that the 4 core algorithm would only outperform the serial algorithm for pulses more than 130 000. For a dataset of 130 000 pulses, the 4 core implementation has, on average, a speedup of 1.02 and the 2 core a speedup of 0.94.

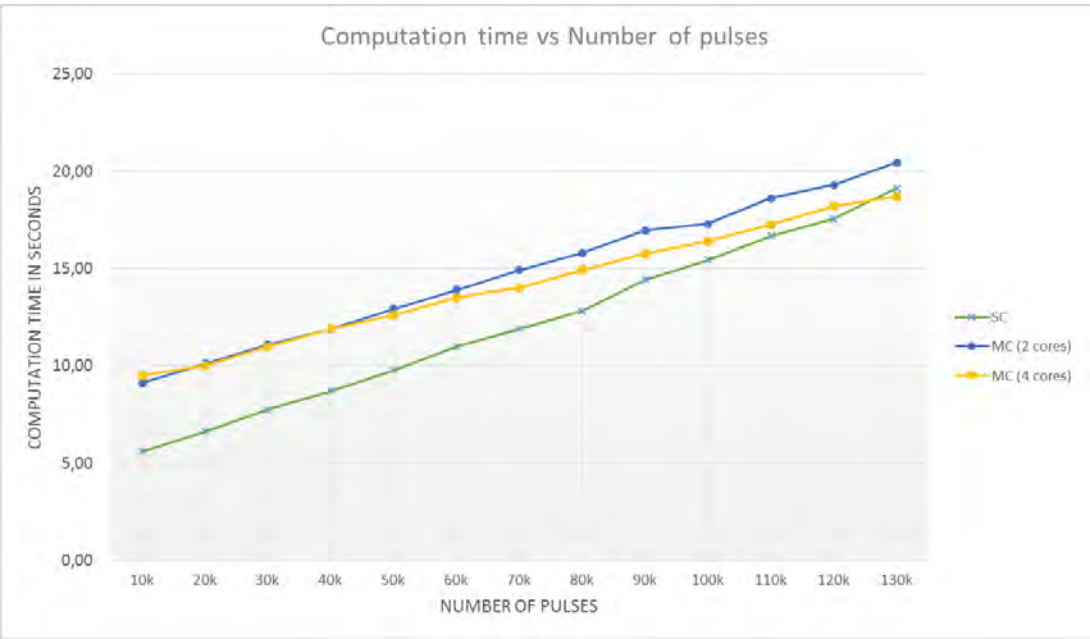


Figure 5.3: Pulse Compression algorithm computation time as a function of pulses processed.

To investigate why the multi-core algorithm was slower than the serial algorithm, a modified version of each algorithm is tested. The computation time of only the correlation operation (that is only the part of the algorithm that can be done in parallel) is considered in each case. Note that it does not consider the time to read in the data or the time to convert a local block of memory into a distributed block of memory. The new computation times are shown in Figure 5.4.

Figure 5.4 shows that when only the correlation was concerned, the 4 core implementation was most efficient when more than 90 000 pulses were processed. For a dataset of 130 000 pulses, the 4 core implementation has on average a speedup of 1.09 and the 2 core has a speedup of 1.01, with respect to the serial algorithm. This suggests that the overhead associated with memory movement between driver and worker processes might be the root of the problem.

The introduction to this chapter mentions how data movement between processes is a significant source of inefficiency. Factors such as declaring distributed arrays, moving data between the driver process and worker processes, the need for more

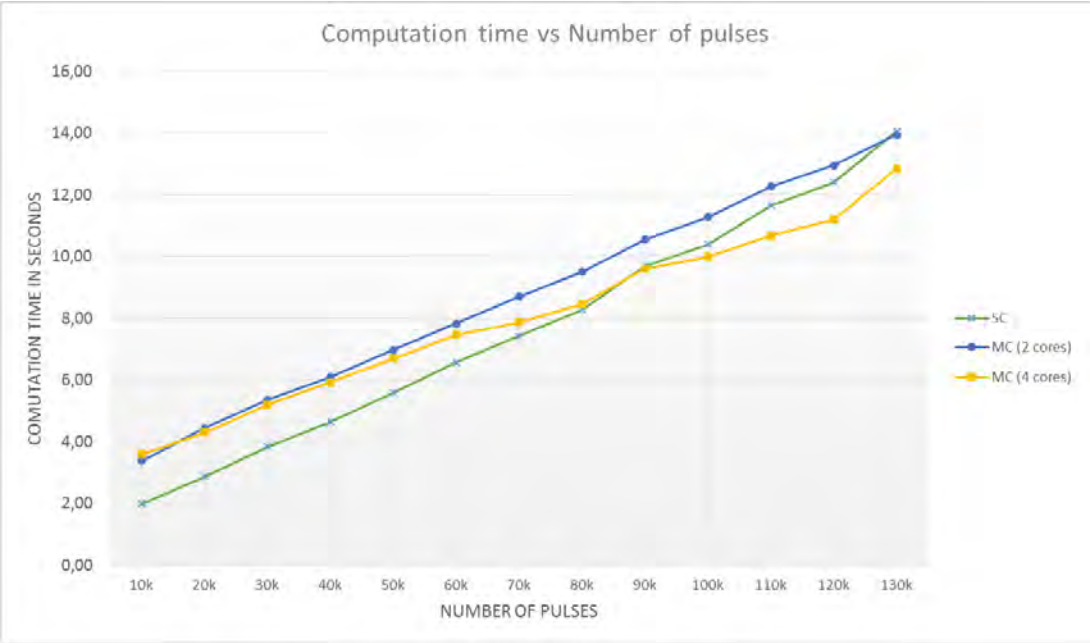


Figure 5.4: Pulse compression algorithm computation time considering only correlation.

variables and functions in the parallel implementation all add to the overhead in Julia's multi-core platform.

5.2 Multi-core Doppler Processing

Doppler processing is done by computing a smaller running discrete time Fourier transform across constant range bins. This commonly requires the pulse compressed data matrix to be transposed. On a multi-core platform this is complicated by the fact that the data is already split into sections in the pulse dimension and distributed across different worker processes. Usually in a GPU implementation, a corner-turning algorithm would be used to reshape the data from each constant range bin. However, due to the way the processed data is split across processes and the slow data movement between processes corner-turning was not implemented.

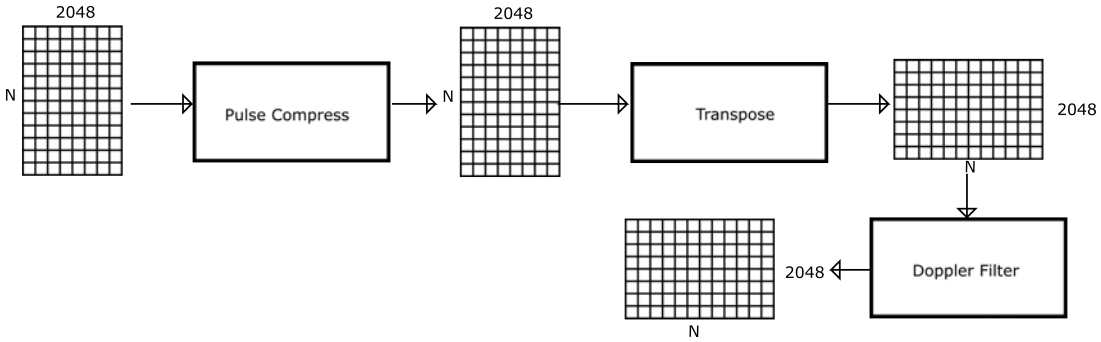


Figure 5.5: A figure illustrating the overall flow of data in the serial Doppler processing algorithm described in NR_DopplerProcessingSC.jl.

Figures 5.5 and 5.6 illustrate the overall flow of the data for serial and a multi-core compression implementation respectively.

5.2.1 Multi-core Doppler Processing Code

Table 5.2 shows the script available for multi-core pulse-Doppler processing.

Table 5.2: Table of available Julia scripts for Doppler processing

Script name	NR_DopplerProcessingMC
Dependancies	NR_PulseCompressionMC, NR_ReadBinFile, NR_ReadRefSig
Function	Applies multi-core pulse compression and pulse-Dopper processing with DFT size as a parameter.

Algorithm Description The script defines a function called `NR_DopplerFilterBank()`, which performs multiple small Fast Fourier Transforms of size `ds` on each constant range bin, effectively applying a bank of Doppler filters to a given array. This function will be evoked on each worker process in parallel. In the function `MCDP()` the multi-core pulse compression routine is called producing pulse compressed data. The distributed data is relocated to a local array and concatenated to form one matrix. Recall that

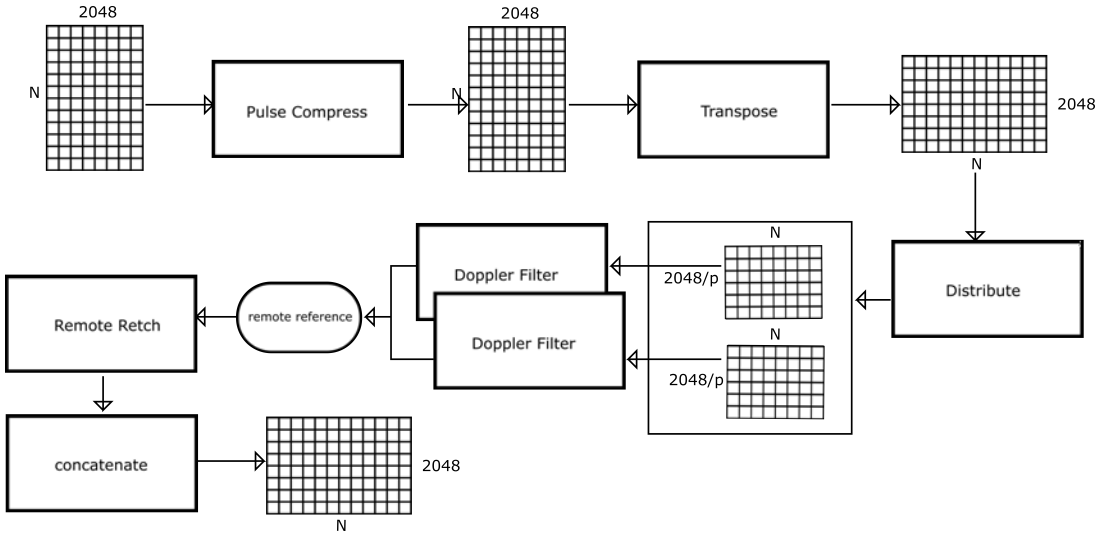


Figure 5.6: A figure illustrating the overall flow of data for a 2 core Doppler processing algorithm described in NR_DopplerProcessingMC.jl.

the matched filtered data has dimensions (slow-time $\times$ fast-time) thus, it is transposed in order to apply the Doppler filterbank in the fast-time dimension. The data is then redistributed in the constant range dimension across the available worker processes. The filterbank is spawned on all available worker processes and is run on each process simultaneously. The remote references to the Doppler processed data is then stored in the array `refs`. The references are then fetched in a parallel map using the `pmap()` function and the result is stored in `DdataArray`.

The algorithm is also expressed in pseudocode in Algorithm 6.

5.2.2 Results and Discussion

Again the computation time of the different algorithms is compared. Figure 5.7 shows graphs of computation time as a function of number of pulses processed. Note that in this case the number of pulses considered were multiples of the DFT size. It compares the serial algorithm with the multi-core algorithm using 2 cores and 4 cores.

Algorithm 6 NR_DopplerProcessingMC: NetRAD Doppler Processing

▷ NR_DopplerFilterBank must be defined @everywhere
function NR_DopplerFilterBank(np_, chunk_, ds_, inputD_)
    define sp ▷ array that holds the chunk of the signal spectrum
    $P \leftarrow$ fft plan on s ▷ find best FFT implementation for data
    for $m \leftarrow 1, chunk_$ **do** ▷ on each range line
        for $n \leftarrow 1 : ds_ : np_$ **do** ▷ for window of size $ds_$
            $s \leftarrow$ fftshift$(pl * (input_[ds_]))$ ▷ apply FFT plan
        end for
    end for
    return s
end function
function MCDP(np_, dftSize_, dFile_, rFile_)
    $PCdata \leftarrow$ pulse compressed from $dFile_$ ▷ see NR_PulseCompressMC
    $Dd \leftarrow PCdata$ converted local array ▷ $PCdata$ is a distributed array
    $p \leftarrow$ number of processes
    $data \leftarrow$ concatenate chunks in Dd
    $data \leftarrow$ transpose $data$
    define $Ddata$ ▷ a distributed array
    $Ddata \leftarrow$ distributed $dRaw$ ▷ distribute the $dRaw$ into P number of chunks along range bins
    define $DPchunk$ ▷ the size of the chunk
    for $w \leftarrow 1, P$ **do**
    ▷ assigns only one chunk to each process
        $refs \leftarrow$ spawn the chunks from $Ddata$ on all processes and perform NR_Dopplerfilterbank, return remote reference
    end for
    define $DdataArray$ ▷ array to hold the pulse compressed chunks of data
    $DdataArray \leftarrow$ fetched data from remote references ▷ uses a parallel map
    return $DdataArray$
end function

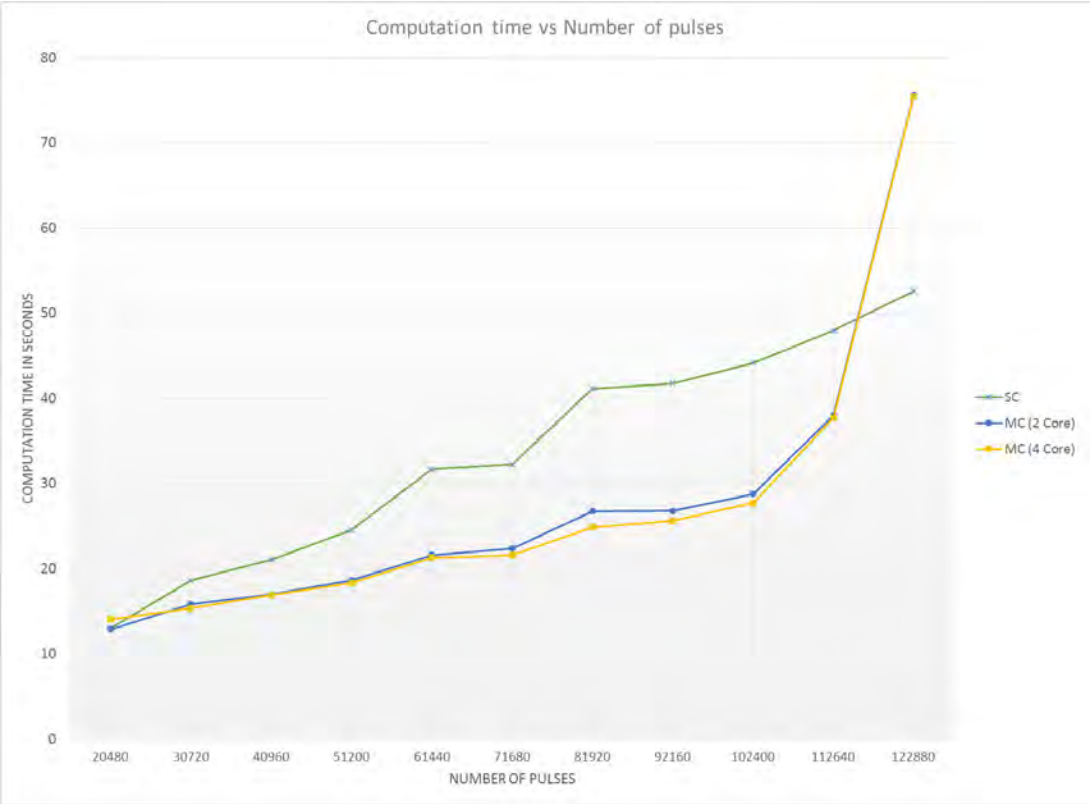


Figure 5.7: Figure comparing the computation times of serial and multi-core Doppler processing algorithms.

The graph shows that the multi-core algorithm outperforms the serial algorithm for pulses in the range $[30\ 720, 112\ 640]$. At best, the multi-core algorithms were found to have speedups of 1.54 (2 core) and 1.60 (4 core) with respect to the serial algorithm occurring at roughly 102 400 pulses. Although the 4 core implementation is faster than the 2 core version in terms of computation time, there was less than a 2 second difference at any point. When processing 112 640 pulses, the multi-core algorithm takes an abnormally long time to run. In both cases the algorithms took roughly 75 seconds. This suggests that they are hitting a bottleneck during runtime. The reason for this bottleneck could be attributed to memory architecture or inter-processor memory movement since all the parameters, other than the number of pulses considered, are kept constant.

The multi-core pulse-Doppler processing algorithm relies on the multi-core pulse

compression, but from the results in Figure 5.3 the serial pulse compression routine runs more efficiently than the multi-core routines for most dataset sizes considered. It would thus be useful to introduce a hybrid algorithm using serial pulse compression combined with distributed Doppler processing. Figure 5.8 considers the computation time of the serial and the hybrid multi-core algorithms.

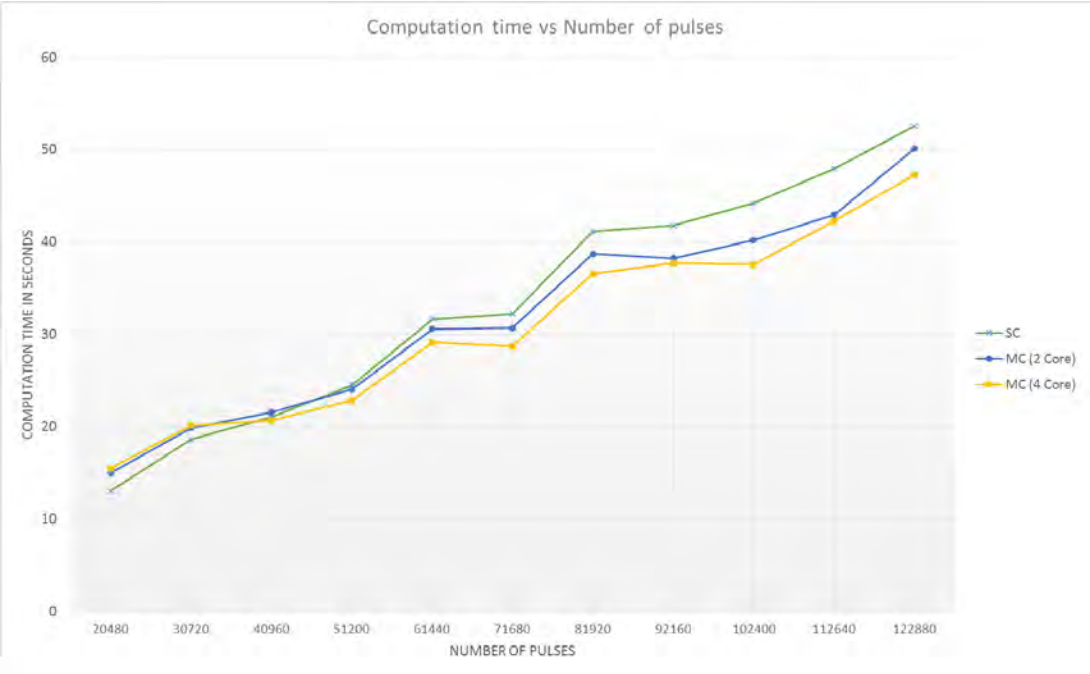


Figure 5.8: Figure comparing the computation times of serial and hybrid multi-core Doppler processing algorithms.

The intersection point in Figure 5.8 shows that the hybrid algorithm becomes more efficient for pulses more than 40 960. With 122 880 pulses considered, the 2 core version has a speedup of 1.04 and the 4 core a speedup of 1.13, both with respect to the serial algorithm.

5.3 Chapter Summary

Julia's parallel platform is based on a unique message passing system. The communication allows the programmer control of only one side in a two-sided

process. Although this makes the implementation simpler, the high-level functions leave the programmer with little control over low-level operations. Sending messages and data movement are the biggest source of overhead in a parallel program. Performing radar signal processing techniques such as pulse compression and pulse-Doppler processing rely heavily on data movement. In both pulse compression and pulse-Doppler processing the idea behind the multi-core algorithms is to exploit the independence of pulses and constant range bins in each technique respectively. That is, the radar data can be split into parts and the processing can be done on the smaller parts.

Theoretically, any parallel program can be split into a purely serial part and a parallel part. Once expressed in this form, Amdahl's law gives the maximum speedup achievable. For pulse compression the results showed that practically the serial version is more efficient in terms of computation time for pulses smaller than 130 000. Further testing showed that by considering only the correlation, the 4 core program was most efficient for datasets which contain more than 90 000 pulses. This highlights the overhead in Julia's multi-core platform.

The Doppler processing algorithm calls the multi-core pulse compression routine. Due to this, it is necessary for the data, which was originally distributed in one dimension, to be collected on the driver process, concatenated, transposed and redistributed in the other dimension. Despite this, the multi-core algorithm performed better than the serial version for pulses more than 20 480. At best, the multi-core algorithms were found to have speedups of 1.54 (2 core) and 1.60 (4 core) with respect to the serial algorithm occurring at roughly 102 400 pulses. After this point, the multi-core algorithm displays an unusual spike in computation time. Since nothing, but the amount of data has increased, this must be due to a memory problem. It is likely that there is a bottleneck in the memory hardware architecture.

Finally, a hybrid algorithm, which makes use of the more efficient serial pulse compression method, is implemented. This method did not result in the abnormality seen before. In this case the multi-core version is 4 seconds faster than the serial version when processing 130 000 pulses as shown in Figure 5.8.

Chapter 6

Conclusions and Recommendations

Altogether this dissertation has set out to develop supporting signal processing scripts which researchers using NetRAD and NeXtRAD data can use to navigate though HDF5 files, produce pulse compressed data, perform pulse-Doppler processing and apply a LMS adaptive filter for WiFi interference suppression.

NetRAD is a project that was formed to capture radar data in a bistatic and multistatic configuration. Its predecessor NeXtRAD, will operate at X- and L-Band while NetRAD operated in S-Band. Hundreds of Gigabytes of NetRAD data already exists online but navigating through the available data for datasets can be a challenging and time consuming task.

In an effort to aid sea clutter research, the infrastructure implemented supports the acquisition and preparation of data in Julia Programming Language.

Julia is an open-source, high level, high performance language, which was designed for scientific programming. Despite being a young language it already promises to be a good alternative to languages such as Python and Matlab.

The contributed support for data acquisition includes a script and documentation in Chapter 3 to navigate through an HDF5 file using Julia to retrieve radar echo

data as well as metadata stored as attributes in the file.

Chapter 4 presented the available scripts for pulse compression, pulse-Doppler processing and interference suppression.

In Section 4.2.2 it highlighted Julia's heavy reliance on virtual memory since memory could not be easily cleared, but shows with enough RAM the pulse compression algorithm runs fast enough for real-time application using only a single core.

It showed the implementation of an adaptive LMS filter for interference suppression and discusses the filter's ability to improve the precision and recall of an OS-CFAR detector. The filter and detector parameters were optimised for one dataset and validated on two others. In pulses with High Clutter contamination, the detector was unable to distinguish between signal and interference, whereas it produced a precision of 100% in pulses with Low Clutter contamination. Even though detection in High Clutter was not achieved the filter was able to reduce the number of highly contaminated pulses.

Chapter 5 made use of Julia's multi-core platform to implement parallel pulse compression and pulse-Doppler processing algorithms. Julia's multi-core platform gives the user control over only one side of a two-sided process and is based on a message passing system. The results highlighted Julia's slow data movement between processes and how it sigificantly influenced the overall speedup achievable.

For future work the pulse compression and Doppler processing scripts should be altered to work in real-time. This would enable users to produce quick-looks during data campaigns to gauge whether the datasets are corrupted by interference and get other relevant information. The Julia Base library is constantly evolving and working on support for GPU processing in the future. It would be worthwhile investigating different types of parallel algorithms and implementations to compare to the ones presented in this project.

Furthermore, the toolbox could always be expanded to include statistical and other post-processing tools relevant to sea clutter analysis. These could

eventually form a comprehensive library for NetRAD and NeXtRAD processing.

Appendix A

Multistatic Radar

A collection of multiple monostatic radar or bistatic radar components which are spatially separated, have overlapping coverage area and work together as one system is loosely referred to as a multistatic, netted or multisite radar system. The main limitation of bistatic radar according to Davies [36] is its limited useful coverage, which is constrained to a vertical fence-like volume in the region of the vertical plane which contains the transmitter and receiver. Many applications require a larger useful coverage.

Since we can have monostatic and bistatic components in a multistatic system the advantages/disadvantages of each transmit and/or receive pair will apply to the whole system [37]. The following characteristics are unique to the multistatic arrangement.

The coverage of the system is limited, but it can be increased by spreading the radar geometry throughout the area to be surveyed [36, 37].

Combining information from multiple transmit and/or receive pairs with shared coverage has benefits. The most obvious advantage is perspective gained from spatial diversity [3] being able to view an area from multiple angles is advantageous when environmental clutter is prominent in any one

transmit-receive pair. That is, by weighting and integrating individual returns, detection can be optimised when more emphasis is placed on stronger returns obtained from certain monostatic or bistatic RCS values. Thus, it is useful when it comes to multipath or shadowing effects that might otherwise lead to poor detection performance, which is an unavoidable consequence of using only one radar. A notable area of interest pertaining to NetRAD is in sea clutter, where the diversity in reflectivity and Doppler shift may prove beneficial for detection in a maritime environment [6].

Usually, a monostatic radar suffers from poor cross-range resolution compared to down-range, so there is potential to exploit the fact that points of interest are in the intersection of constant bistatic range ellipses since there are multiple diverse down-range profiles. This, however, involves a process of associating individual target detections with each other to form a joint detection. Care must be taken with regard to "ghost targets" which can be formed. To decrease these ambiguous results one can use extra information such as Doppler, increase the resolution or add more transmit/receive nodes for angular diversity. A benefit of bistatic systems is the increased range and angular resolution by doing both spatial and time filtering in either case [37].

Target features such as the changes in the RCS can be observed by the transmit/receive pairs in the system. So the gain in information from multiple angles can aid target classification.

Usually, a fault in either transmit or receive in a monostatic/bistatic system can result in a complete loss of functionality. Thus having multiple paired nodes increases the robustness of the system. Quiet receivers and distant transmitters working together may also make nodes difficult to locate and less susceptible to attack in military applications [37].

In order to deduce information such as the range and velocity of targets in the area of interest it is necessary to know the spatial location of the individual radar is the system. They need to share a common timing system

and frequency standard. These must be maintained when the transmitter and receiver lose direct line of sight. It is thus important to have accurate time and phase synchronization between receivers.

Having multiple nodes significantly increases the amount of data that needs to be processed, which in turn can become very computationally complex.

Appendix B

Digital Filters

In digital signal processing we refer to a system, which accepts an input signal, and performs a transformation, which produces a modified output signal, as a digital filter. Linear Shift-Invariant (LSI) systems or filters also known as discrete linear time-invariant systems, are of particular interest. A system is defined as linear if it exhibits homogeneity and superposition [38]. That is, the system is called homogeneous if a scaled version of the input of the filter produces a similarly scaled output. The system exhibits superposition if the output of the sum of two inputs is the same as the sum of the output of the individual inputs. Consider inputs $x_1[n]$ and $x_2[n]$, which when put through a filter $h[n]$, produce outputs $y_1[n]$ and $y_2[n]$, then the system is linear if

$$Ax_1[n] + Bx_2[n] \xrightarrow{h[n]} Ay_1[n] + By_2[n], \qquad \text{A, B} \in \mathbb{R}. \tag{B.1}$$

A digital system is called shift-invariant if a delay or an advance in the input signal causes a similar delay or advance in the output of the system. In other words, the system is LSI if

$$x[n-a] \xrightarrow{h[n]} y[n-a], \qquad \text{for some time } a \in \mathbb{Z}. \tag{B.2}$$

Implementation of Digital Filters In the digital domain, once the filter $h[n]$ is chosen there are some basic methods of implementing it. These include time-domain convolution, frequency domain convolution and time-domain difference equations. These are outlined in [38] and are summarised below. Time-domain convolution requires knowledge of the impulse response $h[n]$ of the LSI system, and the output is basically the sum of a point-wise product of the signal $x[n]$ with the flipped version of the impulse response or vice-versa. Mathematically we can write

$$y[n] = \sum_{m=-\infty}^{\infty} x[m]h[n-m] = \sum_{m=-\infty}^{\infty} h[m]x[n-m]. \tag{B.3}$$

In radar both the impulse response and the signal are often stored as complex-valued sample points, so care must be taken to use "complex" multiplication and accumulator operations. Note that this form of convolution is only practical for FIR filters (i.e filters of a finite length).

Frequency-domain convolution makes use of the relationship shown in (2.3), we are familiar with continuous time-domain convolution being equivalent to continuous frequency domain multiplication via the convolution theorem of Fourier transforms. However, (2.3) relies on the DTFT, which is not a practically computable transform, since it is a function of the continuous variable (i.e. $\hat{f}$), thus practically we rely on the discrete DFT instead. For a K-point DFT, the process of filtering by multiplying two DFTs is known as frequency-domain convolution. The DFT of the filter is,

$$Y[k] = H[k]X[k], \qquad k = \{0, 1, ..., K-1\}. \tag{B.4}$$

Recall that to create the DFT we are effectively sampling the DTFT of the continuous input signal. Thus, we have in effect created a periodic replication of the original input signal. As a result we have to use circular convolution as opposed to the standard convolution technique, which can often be the cause of error. However the standard convolution method can also be used if K, the DFT size, is chosen appropriately. This frequency convolution technique is often

also referred to as fast convolution, however, it is not necessarily faster than the direct method of time-domain convolution as it is completely dependent on the parameters [38].

Finally, when dealing with infinite impulse response (IIR) filters it is necessary to express the system transfer function, in the z-domain such that we have

$$Y(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 + \sum_{k=1}^{N} a_k z^{-k}} X(z), \qquad a_k, b_k \in \mathbb{R}. \tag{B.5}$$

Where a_k and b_k follow from the poles and zeros in the design of the IIR filter. Then rearranging and applying the z-transform property we can obtain a difference equation to compute the filter output $y[n]$ as

$$y[n] = \sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k]. \tag{B.6}$$

A more detailed account of the design process can be found in [39].

Appendix C

Julia for Ne(X)tRad Manual

The following chapter is a manual to aid the user with using Julia for NetRAD/NeXtRAD development. All the scripts are available on the accompanying CD as well as on github.

C.1 Getting Started

The first step to doing analysis is to acquire the data from the relevant database. Once the HDF5 file names have been acquired the next step is to feed the data into the Julia environment.

Starting Julia Once downloaded Julia can be run in the Linux terminal. When in the terminal, navigate to the directory which holds the HDF5 files and start Julia using the command:

```
julia
```

To start Julia on four cores use:

```
julia -p 4
```

Make sure that you are running at least v0.4.5 for all the functions to work properly. Julia is constantly evolving and some scripts might need altering if code from Julia's Base library is changed.

Loading HDF5 files Once in the Julia environment, the data stored in (*.h5) files can be read in using the HDF5.jl script with `include("NR_HDF5.jl")`. The raw data will be read into a Julia array.

C.2 Data Preparation

The raw NetRAD data is stored directly from the ADC as 16-bit unsigned integers. The ADCs can only store positive numbers thus a DC offset was introduced to make capture of negative samples possible. For this reason there is an offset that must be removed and the new values need to be converted to voltages. Table C.1 shows the scripts to read in radar data from binary files and convert the raw data values to real data values as well as a script to read in the reference signal from a text file and process it.

If you prefer to use the available scripts to perform pulse compression and Doppler processing then use `include("NR_PulseCompressSC.jl")` and `include("NR_DopplerProcessingSC.jl")` respectively to produce match filtered data and Doppler processed data.

C.3 Quickplots

To create a snapshot Quickplot of Doppler processed data, include the PyPlot library, plot the data and save plot to file:

```
using PyPlot
```

```
# The DFT size must be defined and the same as the one used
# for Doppler processing
dftsize = 255;

# Plot one cut
image = imshow(log10(abs(s[:,1:1+dftsize])), aspect = "auto");

# Save the png to file
using PyCall
@pyimport matplotlib.pyplot as plt

plt[:savefig]("range_doppler.png");
```

C.4 Saving to binary files

If you would like to save a data array to a file after multi-core pulse compression you can write to a file in this way:

```
myFile = open("NR_PCdata.bin", "w+");

# P is the number of cores used
for w = 1:P
# DdataArray is an array containing the processed data
# from all processors
write(myFile, DdataArray[w][1]);
end
```

Table C.1: Table of available Julia scripts for raw data processing

Script name	NR_ReadRefSig.jl
Dependancies	None
Function	Reads in the reference signal from a text (*.txt) file. The values in the text file are assumed to be comma separated values. The function returns `RefSig` which is a 1 x 2048 complex conjugated, Fourier transformed reference signal array, which has been normalised, windowed (Hanning) and zero-filled.
Script name	NR_ReadBinFile.jl
Dependancies	None
Function	Reads in the raw radar echo data from a binary (*.bin) file. Each raw pulse from the binary file is assumed to be a 16-bit array of unsigned integers. The number of pulses can be specified by choosing N. The bin file data is assumed to be a 2048Nx1 array. The function returns (N x 2048) 16-bit array.

Appendix D

Useful HDF5 Object Operations

An outline of some of the operations that can be performed on files and objects are listed in the tables below according to the HDF5 1.6 Standard.

Table D.1: Table of basic operations performed on a file

Operation	**Description**
Create	Creates a file with defined structural metadata elements.
Open	Opens a file.
Reopen	Reopens a previously opened file.
Close	Closes a file.
Mount/Unmount	Includes/removes one files's graph structure in/from another file's graph structure.
Get properties	Gets the structural metadata of the file.
IS HDF?	Queries whether a file is an HDF5 file.

Table D.2: Table of basic operations performed on a group

Operation	**Description**
Create	Creates a group with specified properties.
Open	Opens an existing group.
Close	Closes a group.
Iterate	Iterates through group members.
Get information	Gets the number of group members; a member's object type; and the name of a link that points to a member.

Table D.3: Basic operations performed on a link

Operation	**Description**
Insert	Inserts a link into a group.
Remove	Removes a link from a group.

Appendix E

Hardware Specifications

Table E.1: Hardware specifications of the server used for computation

Name	**Description**
Server Name	gpuserv
Number of CPUs	8
CPU Model	Intel(R) Core (TM) i7 960 @ 3.20GHz
CPU Architecture	x86_64
CPU MHz	1600
RAM	23GB
Disk Memory	55.9GB + 8GB SWAP

Bibliography

[1] R. T. Lord and M. Inggs, "Approaches to RF interference suppression for VHF/UHF synthetic aperture radar," Tech. Rep.

[2] N. J. Willis, "Bistatic Radar," in *Radar Handbook*, 3rd ed., M. Skolnik, Ed. McGraw-Hill, 2008, ch. 23.

[3] T. Derham, S. Doughty, K. Woodbridge, and C. Baker, "Design and evaluation of a low-cost multistatic netted radar system," *IET Radar Sonar Navig.*, vol. 1, no. 5, pp. 362–368, 2007. [Online]. Available: http://eprints.ucl.ac.uk/13289/

[4] M. Inggs, A. Balleri, W. A. Al-Ashwal, K. D. Ward, K. Woodbridge, M. Ritchie, W. Miceli, R. J. A. Tough, C. J. Baker, S. Watts, R. Harmanny, A. Stove, J. S. Sandenbergh, and H. D. Griffiths, "NetRAD multistatic sea clutter database," *International Geoscience and Remote Sensing Symposium (IGARSS)*, pp. 2937–2940, 2012.

[5] M. Inggs, S. Coetzee, H. Griffiths, F. Fioranelli, M. Ritchie, and K. Woodbridge, "Database Design for an Experimental, Dual Band, Polarmetric Radar," *IEEE*, pp. 417–421, 2015.

[6] M. Inggs, G. Inggs, and S. Sandenbergh, "Multistatic Networked Radar for Sea Clutter Measurements," pp. 4449–4452, 2011.

[7] S. Watts, "Radar Sea Clutter Modelling and Simulation - Recent Progress and Future Challenges," *Proceedings IET Seminar on Radar Clutter Modelling*, pp. 1–7, 2008.

[8] ——, "Radar sea clutter: Recent progress and future challenges," *2008 International Conference on Radar*, pp. 10–16, 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4653882

[9] W. Al-Ashwal, C. J. Baker, A. Balleri, H. D. Griffiths, R. Harmanny, M. Inggs, W. J. Miceli, M. Ritchie, J. S. Sandenbergh, A. Stove, R. J. Tough, K. D. Ward, S. Watts, and K. Woodbridge, "Statistical analysis of simultaneous monostatic and bistatic sea clutter at low grazing angles," vol. 47, no. 10, 2011. [Online]. Available: http://discovery.ucl.ac.uk/1307014/

[10] K. D. Ward, R. J. a. Tough, and P. W. Shepherd, "Modelling Sea Clutter: Correlation, Resolution and Non-Gaussian Statistics," *International Conference on Radar*, no. 449, pp. 95–99, 1997.

[11] M. R. Ducoff and B. W. Tietjen, "Pulse Compression Radar," in *Radar Handbook*, 3rd ed., M. Skolnik, Ed. McGraw-Hill, 2008, ch. 8.

[12] K. Ounissi, "Advanced Signal Processing Techniques for Puluse-Doppler Radar," Master of Science, Chalmers University of Technology, 2006.

[13] Stimson, "Pulse Compression and High-Resolution Radar," 2014, ch. 16, pp. 233–248.

[14] M. A. Richards, J. A. Scheer, W. A. Holm, and W. L. Melvin, *Principles of Modern Radar - Basic Principles*. SciTech Pub., 2010, vol. 1.

[15] M. A. Richards, "Doppler Processing," in *Principles of Modern Radar - Basic Principles*, vol. 1, ch. 17.

[16] R. M. O. Donnell, "Radar Systems Engineering Lecture 13 Clutter Rejection Part 2 - Doppler Filtering Block Diagram of Radar System," *Systems Engineering*, pp. 1–67, 2009.

[17] S. M. Kay, *Fundamentals of statistical signal processing: estimation theory*, A. V. Oppenheim, Ed. University of Rhode Island: Prentice Hall PTR.

[18] A. Feuer and E. Weinstein, "Convergence Analysis of LMS Filters with Uncorrelated Gaussian Data," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 33, no. 1, pp. 222–230, 1985.

[19] V. W. Lee, P. Hammarlund, R. Singhal, P. Dubey, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, and S. Chennupaty, "Debunking the 100X GPU vs. CPU myth," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, p. 451, 2010.

[20] C. A. Lee, S. D. Gasster, A. Plaza, C.-I. Chang, and B. Huang, "Recent Developments in High Performance Computing for Remote Sensing: A Review," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 4, no. 3, pp. 508–527, 2011.

[21] L. Chen, Z. Hu, J. Lin, and G. R. Gao, "Optimizing the fast fourier transform on a multi-core architecture," *Proceedings - 21st International Parallel and Distributed Processing Symposium, IPDPS 2007; Abstracts and CD-ROM*, pp. 1–8, 2007.

[22] C. Gu, W. Chang, X. Li, and Z. Liu, "Multi-Core DSP Based Parallel Architecture for FMCW SAR Real-Time Imaging," *Radioengineering*, vol. 24, no. 4, pp. 1084–1090, 2015.

[23] J. Bezanson, S. Karpinski, V. Shah, and A. Edelman, "Why we created Julia." [Online]. Available: http://julialang.org/blog/2012/02/why-we-created-julia

[24] "Julia High-Performance JIT Compiler." [Online]. Available: http://julialang.org/

[25] M. Nolta and et. al., "Types," 2012. [Online]. Available: http://docs.julialang.org/en/release-0.4/manual/types/

[26] M. Folk and E. Pourmal, "HDF5 Data Model, File Format and Library - HDF5 1.6," Tech. Rep., 2007.

[27] K. Hinsen, T. Holy, and S. Kornblith, "Julia HDF5 Guide," 2015. [Online]. Available: https://github.com/JuliaLang/HDF5.jl/blob/master/doc/hdf5.md

[28] "Julia Docs, Mathematics, Signal Processing." [Online]. Available: http://docs.julialang.org/en/release-0.5/stdlib/math/?highlight=fftw

[29] M. Ritchie, F. Fioranelli, H. Griffiths, and B. Torvik, "Micro-drone rcs analysis," in *2015 IEEE Radar Conference*, Oct 2015, pp. 452–456.

[30] F. Fioranelli, M. Ritchie, and H. Griffiths, "Multistatic human micro-doppler classification of armed/unarmed personnel," *IET Radar, Sonar Navigation*, vol. 9, no. 7, pp. 857–865, 2015.

[31] J. L. Gustafson, "Reevaluating Amdahl's Law," vol. 31, no. 5, pp. 532–533, 1988.

[32] D. Wang, M. Ali, and E. Blinka, "Synthetic Aperture Radar (SAR) Implementation on a TMS320C6678 Multicore DSP," pp. 1–9, jan 2015.

[33] "Julia Docs, Parallel Computing." [Online]. Available: http://docs.julialang.org/en/release-0.4/manual/parallel-computing/

[34] N. Bershad, J. Bermudez, and J.-Y. Tourneret, "An Affine Combination of Two LMS Adaptive Filters—Transient Mean-Square Analysis," *IEEE Transactions on Signal Processing*, vol. 56, no. 5, pp. 1853–1864, 2008.

[35] J. Bolewski, K. Kamieniecki, T. Besard, T. Kelman, V. Churavy, and S. Danisch, "JuliaGPU." [Online]. Available: https://github.com/JuliaGPU

[36] D. E. N. Davies, "Use of bistatic radar techniques to improve resolution in the vertical plane," *Electronics Letters*, vol. 4, no. 9, pp. 170–171, 1968. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4210077

[37] E. Hanle, "Survey of bistatic and multistatic radar," *IEE Proceedings (Communications, Radar and Signal Processing)*, vol. 133, no. 7, pp. 587–595, 1986.

[38] M. A. Richards, "Digital Signal Processing Fundamentals for Radar," in *Principles of Modern Radar - Basic Principles*, M. A. Richards, J. A. Scheer, and H. W. A, Eds. SciTech Pub., 2010, vol. 1, ch. 14.

[39] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-Time Signal Processing*, 2nd ed. Prentice-Hall, Inc, 1999, vol. 23, no. 2.