IMPLEMENTATION AND PERFORMANCE OF FACTORIZED BACKPROJECTION ON LOW-COST COMMERCIAL-OFF-THE-SHELF HARDWARE

THESIS

Alec S. Rasmussen, Capt, USAF

AFIT-ENG-MS-16-M-041

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-16-M-041

IMPLEMENTATION AND PERFORMANCE OF FACTORIZED BACKPROJECTION ON LOW-COST COMMERCIAL-OFF-THE-SHELF HARDWARE

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

Alec S. Rasmussen, B.S.

Capt, USAF

March 2016

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-MS-16-M-041

IMPLEMENTATION AND PERFORMANCE OF FACTORIZED BACKPROJECTION ON LOW-COST COMMERCIAL-OFF-THE-SHELF HARDWARE

THESIS

Alec S. Rasmussen, B.S.
Capt, USAF

Committee Membership:

Dr. Julie Jackson
Chair

Dr. Peter Collins
Member

Dr. Richard Martin
Member

AFIT-ENG-MS-16-M-041

Abstract

Traditional Synthetic Aperture Radar (SAR) systems are large, complex, and expensive platforms that require significant resources to operate. The size and cost of the platforms limits the potential uses of SAR to strategic level intelligence gathering or large budget research efforts. The purpose of this thesis is to implement the factorized backprojection SAR image processing algorithm in the C++ programming language and test the code's performance on a low cost, low size, weight, and power (SWAP) computer : a Raspberry Pi Model B. For a comparison of performance, a baseline implementation of filtered backprojection is adapted to C++ from pre-existing `MATLAB`® code. The factorized backprojection algorithm shows a computational improvement factor of 2-3 compared to filtered backprojection. Execution on a single Raspberry Pi is too slow for real-time imaging. However, factorized backprojection is easily parallelized, and we include a discussion of parallel implementation across multiple Pis.

Acknowledgements

First and foremost, I want to thank my wife. She has been my muse, my coach, my task master, and my inspiration. Without her, I would never have survived this program. I also want to thank my wonderful daughter, who has been a shining beacon whenever things were dark. My advisor, Dr. Jackson, for her guidance and assistance in deciphering the factorized backprojection literature. And finally, to all my fellow members of class 16M for their support and encouragement as we approached the end of our tenure at AFIT.

Alec S. Rasmussen

Table of Contents

Page

Abstract iv

Acknowledgements v

List of Figures viii

List of Tables x

I. Introduction 1

1.1 Problem Description 1
1.2 Algorithm Usage 2
1.3 Tactical Imagery 3
1.4 Thesis Layout 4

II. Theory and Background 7

2.1 Synthetic Aperture Radar 7
2.1.1 Spotlight SAR Geometry and Data Domains 8
2.1.2 SAR Resolution 12
2.1.3 Sampling Theory 14
2.2 Image Formation Algorithms 15
2.2.1 Tomographic Imaging Methods 17
2.2.2 Factorized Back Projection 19
2.3 Hardware 24

III. Code Development and Methodology 27

3.1 Language Choice 27
3.1.1 Memory Usage 27
3.1.2 Speed of Execution 29
3.1.3 Compatibility 29
3.2 Filtered Backprojection Algorithm Development 31
3.3 Performance of Filtered Backprojection Implemented in C++ vs. MATLAB® 36
3.4 Factorized Backprojection Algorithm Development 44
3.5 Performance of Factorized Backprojection in C++ vs. Filtered Backprojection in C++ 46

IV. Implementation on Raspberry Pi 57

V. Hardware Implementation Discussion 61

Page

VI. Conclusions/Future Work 63

Bibliography 66

List of Figures

Figure Page

1 Radar system block diagram. 2

2 Spotlight SAR data collection scene. 9

3 Image partitioning example using two levels of recursion. Red lines indicate quadrant partitions in the first level of recursion; blue lines depict quadrant partitions in the second level. 20

4 Example of 5-sample anti-aliasing filter applied to phase history data. 24

5 Flow chart for main program block of filtered backprojection. The backprojection subroutine is depicted in Figure 6. 34

6 Flow chart for sub-routines of filtered backprojection, i.e. FFT procedures and pixel summation. 35

7 Location of targets in simulation scene. All the targets have unit amplitude. Targets at $(x, y) = (2.5, -2.5)$, $(2.5, -7.5)$, $(7.5, -2.5)$ and $(7.5, -7.5)$ are located off the (x, y) plane and appear at projected locations in the formed image. 37

8 Zoom in of point scatterer to show resolution cell size. 39

9 Comparison of `MATLAB`® generated image and C++ generated image for the monostatic case. 41

10 Comparison of `MATLAB`® generated image and C++ generated image for the bistatic case. 42

11 Flow chart for main program block of factorized backprojection. 45

12 Time to compute vs. number of recursion levels for the example in Figure 9(b). 48

13 Effect of aliasing on image quality for various levels of recursion. 49

Figure Page

14 Aliasing error introduced through various levels of recursion. Zero levels of recursion is equivalent to filtered backprojection (no aliasing), and has the same error as in Figure 9(c). 50

15 MSE of the aliasing error introduced by recursion. 51

16 Images generated using 2 levels of recursion and varied filter length. 53

17 Mean squared error of the difference between filtered backprojection and 2-level recursion factorized backprojection vs. filter length. 54

18 Memory usage versus levels of recursion for factorized backprojection. 55

19 Time to compute vs. number of recursion levels on Raspberry Pi. 58

20 Comparison of images generated on Development laptop vs. Raspberry Pi for zero recursion levels. 59

21 Comparison of images generated on Development laptop vs. Raspberry Pi for one recursion level. 59

22 Comparison of images generated on Development laptop vs. Raspberry Pi for two recursion levels. 60

23 Comparison of images generated on Development laptop vs. Raspberry Pi for three recursion levels. 60

List of Tables

Table Page

1 Table of Image Geometry Symbols for Figure 2 9

2 Table of Parameters for Monostatic Simulation 38

3 Table of Parameters for Bistatic Simulation 38

4 C++ resource usage versus `MATLAB`® 44

IMPLEMENTATION AND PERFORMANCE
OF FACTORIZED BACKPROJECTION
ON LOW-COST COMMERCIAL-OFF-THE-SHELF HARDWARE

I. Introduction

1.1 Problem Description

Since the 1950's, the United States military has used synthetic aperture radar (SAR) technology extensively to create high-resolution intelligence imaging [1]. SAR imaging operates on a large platform that provides analysts with large scale observations of interest targets. Since the introduction of SAR, the military has continued to demand larger scenes with more detail and higher resolution, requiring more expensive and powerful computers and platforms to create such imagery. However, this begs the questions: Do bigger, better, more complex technologies always serve practical needs? While many scholars show the benefits of large scale SAR imaging [2–6], an increased interest in tactical imaging has opened a niche within the SAR field, specifically at the Air Force Institute of Technology (AFIT) and the U.S. military more generally, that has not been fully explored. With the increase in budget-constrained, time-essential, and localized missions, using SAR on a low-cost, low-size weight and power (SWAP) platform may serve as a very useful tool to add to AFIT and the Air Force's SAR toolkit. This thesis will attempt to research and evaluate the feasibility of processing received SAR data on a low-cost, low-SWAP computer (the signal processor in Figure 1). As there is limited research in the use of small-platform SAR imagery, this project serves as the baseline for future research.

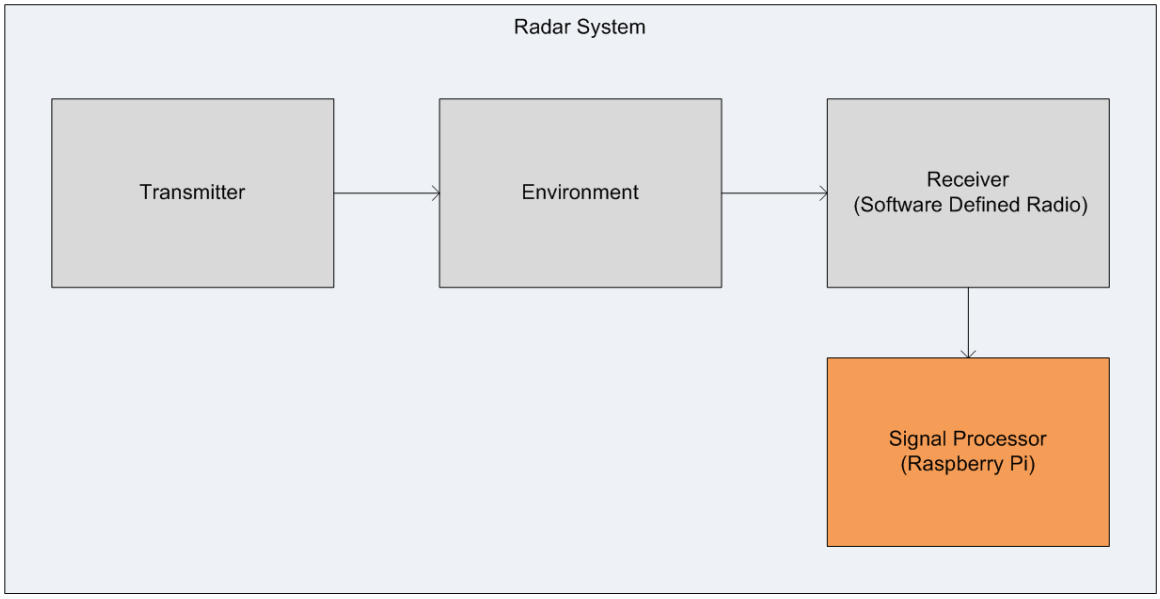


Figure 1. Radar system block diagram.

The research will be conducted in three stages, focusing on the adaptation of SAR algorithms to a small-platform friendly programming language. The project will then focus on adapting the code using factorized backprojection to aid in areas of small-platform technology that are crucial to its operation: reduced memory usage and maintenance of image quality while decreasing run-time. The final stage of this project will then take a functioning SAR algorithm and actually test its ability to run on a small-platform machine. This research opens the door to numerous continuing research endeavors for AFIT while also creating the opportunity for a new use of SAR imaging that could be incorporated into a real-time, low-cost, option for tactical intelligence in localized missions.

1.2 Algorithm Usage

Many algorithms have been developed to process the received radar data into images [2,3]. These algorithms can largely be broken up into two groups: direct Fourier methods and backprojection. In the early 1980's, Munson et al. [5] recognized the sim-

ilarities between spotlight mode SAR and Computerized Axial Tomography (CAT) and proposed a modified version of the convolution backprojection algorithm being used for CAT at the time. Backprojection provides a robust imaging algorithm, but comes with a high computational burden. The high computational burden limits the use of backprojection to expensive, robust workstations or compute clusters. The direct Fourier algorithms, such as the polar format algorithm, reduce the computational burden in exchange for increased active memory usage. For real-time purposes, backprojection operates on a single received radar pulse at a time and can iteratively build up the image with each pulse. The direct Fourier methods require the entire data set to be processed as a whole.

Standard backprojection accounts for non-linear flight paths and has fewer image artifacts caused by large apertures [7]. The major drawback to the convolution backprojection algorithm is that it requires $\mathcal{O}(N^3)$ operations to generate an image as compared to the $\mathcal{O}(N^2 logN)$ operations for the direct Fourier based methods [5]. In [8–11] faster implementations of the backprojection algorithm are introduced. These implementations break up the imaging process into smaller pieces, referred to as factorization. With a less computationally burdensome backprojection algorithm available, cheaper and less powerful hardware can be used.

1.3 Tactical Imagery

The improvements offered by the fast backprojection algorithm come from a reduction in the number of received pulses that are backprojected into the image. The total operation count can be reduced to be on par with the direct Fourier based methods by recursively partitioning the backprojection integral. The partitioning involves breaking the image into smaller subimages and reducing the number of pulses used to create the subimages. This reduction in computational load while providing most

of the benefits of full convolution backprojection has led to the fast backprojection algorithms becoming increasingly popular [4, 7–9, 12–18].

However, the improvements to the fast backprojection algorithm haven't led to a significant increase in the efforts to miniaturize a radar system. Projects such as MicroASAR, NuSAR, and SlimSAR, have worked to reduce the size of radar systems, but these projects use custom hardware and Field Programmable Gate Arrays (FPGA's) to store a digitized demodulated signal locally (on-board the platform), off-load the signal via tactical data link, or process the image locally [19]. Using custom hardware can improve processing efficiency, though often this improvement comes at the cost of increased implementation complexity and/or decreased flexibility of the processor as well as being more expensive.

Using an FPGA provides a programmable, relatively cheap, dedicated processing unit. FPGA's are low power, robust and in a lot of ways an excellent solution to providing a dedicated image processing unit. However, at the time of this thesis the development platform available was a Raspberry Pi (R-Pi). Whereas a FPGA allows for direct programming of the logic circuits to acheive a particular function, the Raspberry Pi is a general purpose microprocessor. An advantage of using a Raspberry Pi is that the Raspberry Pi is significantly cheaper than most development platforms (R-Pi $25, FPGA $89). Another advantage of using the R-Pi is the choice of programming language: FPGA's are programmed in Verilog or VHDL, and R-Pi can pick from many different languages ranging from C to Java to Python.

1.4 Thesis Layout

The overall goal of project is to create a form of the algorithm that maintains high quality imagery and that is time-sensitive with low-memory overhead and low cost. The project will unfold in three main stages: algorithm adaptation, algorithm

revision, and algorithm practical performance.

Chapter 2 will provide a more detailed explanation to the use of SAR imaging, its data collection process, and image creation. Chapter 2 will use a compilation of understanding of Fourier and sampling theory as well as the usage of both filtered and factorized backprojection.

Using Chapter 2 as the overall knowledge base, Chapter 3 will first introduce the criteria used in determining language choice for algorithm adaptation, use of operating system, and an introduction to the experimental hardware platform. Focusing on memory usage, run-time, and quality of imagery maintained, each of the above experimental pieces will be evaluated and determined. Chapter 3 will also serve as the controlled testing for stages one and two of the project. Stage one will evaluate the new code adaptation with the original algorithm used in MATLAB, comparing and contrasting the image quality, memory usage and time to generate an image. Stage two will then implement factorized backprojection and analyze its capability to decrease run time while maintaining image quality. Stage two will also introduce the usage of filters for this very reason and will analyze filter length in relation to effects on run-time.

Chapter 4 evaluate potential transition of the project from the lab into the real-world, testing the actual ability of the script to run on a low-cost, low-SWAP platform. Chapter 4 will analyze the actual feasibility of the adapted script to run on the smaller platform while also evaluating run-time, image quality, and memory usage and noting areas of possible improvement.

Chapter 5 will serve as the central discussion for what this thesis accomplished through the three stages of research, with the goal of creating an initial implementation of factorized backprojection code that has the capability to function on a small platform. Areas of improvement including code implementation, memory usage, pro-

cessor speed, and run-time will all be discussed as possible areas for improvement and future research.

II. Theory and Background

The goal of this chapter is to provide some background on the concepts that will be explored in this thesis. The foundation of the thesis is synthetic aperture radar (SAR) imaging. To generate images through SAR, many different approaches can be taken, mostly revolving around how the received radar signal is collected and processed. Many different factors, ranging from the flight path of the radar system to the imaging algorithm used to generate the final image, affect each other and work together to create an image. As an example, the type of hardware used to process the received data into an image affects what algorithms can be used.

This chapter will discuss the basics of SAR, provide a brief survey of sampling theory and how it applies to SAR, a brief overview of the different imaging algorithms available, and a more in-depth discussion of the convolution/filtered backprojection algorithm. It will also discuss the current state of low cost commercial-off-the-shelf (COTS) hardware being explored in this thesis.

2.1 Synthetic Aperture Radar

Synthetic aperture radar (SAR) is a method of radio detection and ranging (radar) wherein multiple radar pulses are collected from many different look angles of a scene or target. Combining data from the many look angles gives the same effect as having a larger antenna, or aperture. The look angles collected are determined by the direction the radar antenna is pointing, and the flight path of the platform the radar system is on.

From an imaging perspective, the objective is to create a representation of the reflectivity function $g(x, y, z)$ of the scene. If the scene is assumed to be represented by a collection of point scatterers, and assuming that linear superposition is applicable,

then the reflectivity function is the summation of the radar reflectivity of all the scatterers in the scene. Further, assuming that the point scatterers are isotropic the reflectivity function is represented by

$$g(x, y, z) = \Sigma_i A_i \delta(x - x_i, y - y_i, z - z_i) \tag{1}$$

where A_i is the complex amplitude of the reflection from scatterer i and (x_i, y_i, z_i) is the location of scatterer i in the scene [20].

The received and transmitted radar pulses are known; however, the reflectivity function $g(x, y, z)$ itself is not directly known. The received pulse $r(t)$ is a scaled and shifted copy of the transmitted signal $s(t)$. The received pulse $r(t)$ can be written as the convolution of the transmitted signal and the the scene reflectivity function: $r(t) = s(t) * g(t)$ where $*$ is the convolution operator [3]. But the actual scene is $g(x, y, z)$, so what is the relation between $g(x, y, z)$ and $g(t)$? The following discussion summarizes SAR geometry and data relations from [3, 20, 21].

2.1.1 Spotlight SAR Geometry and Data Domains.

Figure 2 represents a generic data collection scene. As a receiving radar platform travels along a path, pulses are received at (ideally) contiguous and evenly spaced azimuth angles θ_{Rx} while the grazing angle ψ_{Rx} is kept constant. The same is kept true for the transmitter, with the difference that pulses are transmitted instead of received. The bistatic angle β is the angle between the transmitter and receiver lines of sight. See Table 1 for a description of the image geometry symbols.

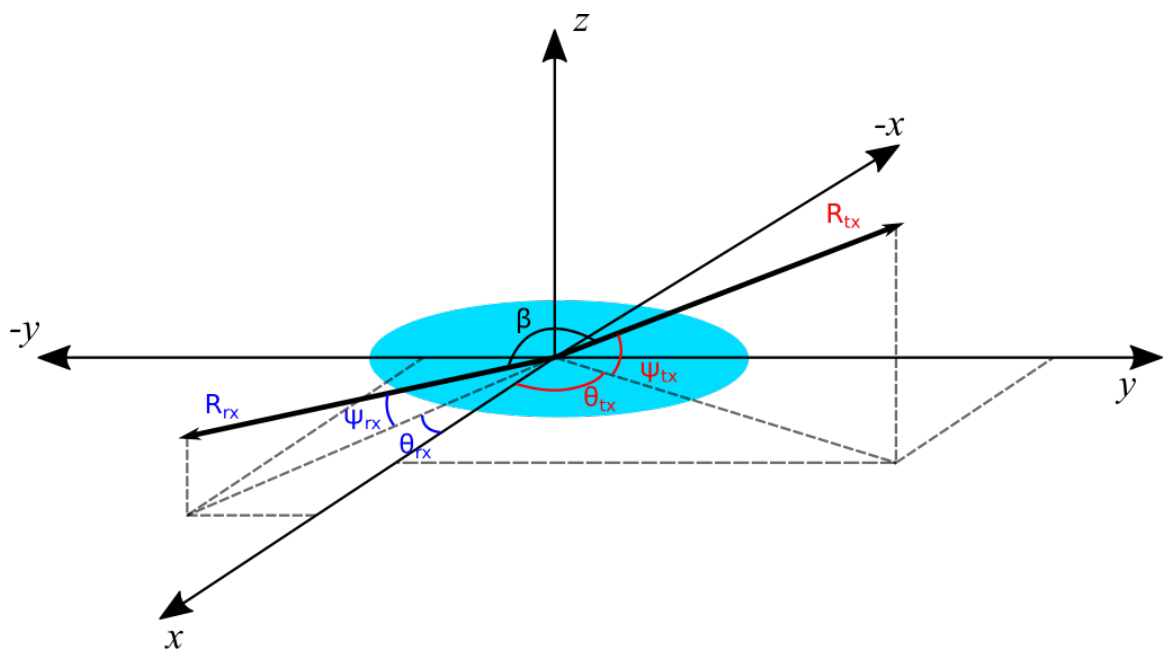


Figure 2. Spotlight SAR data collection scene.

Table 1. Table of Image Geometry Symbols for Figure 2

Variable	Description
ψ_{Tx}	the angle at which the transmitter pulse illuminates the target scene (grazing angle)
θ_{Tx}	the transmitter azimuth angle from scene reference coordinate system
ψ_{Rx}	the grazing angle of the receiver
θ_{Rx}	the receiver azimuth angle from the scene reference coordinate system
β	the angle between transmit and receive look angles
R_{Tx}	the range from the transmitter to a point in the scene
R_{Rx}	the range from the receiver to a point in the scene

Since (x, y, z) are position coordinates, time can be converted to a distance u by multiplying by the speed of light. Radar pulses propagate spherically, and the imaged scene is represented in Cartesian coordinates, so u and (x, y, z) share a spherical-Cartesian relation. The spherical to Cartesian coordinate conversion is

$$\begin{aligned} x =& u \cos\theta \cos\psi \\ y =& u \sin\theta \cos\psi \\ z =& u \sin\psi \end{aligned}$$

and the Cartesian to spherical coordinate conversion is

$$\begin{aligned} u =& \sqrt{x^2 + y^2 + z^2} \\ \theta =& \tan^{-1}\left(\frac{y}{x}\right) \\ \psi =& \cos^{-1}\left(\frac{z}{\sqrt{x^2 + y^2 + z^2}}\right) \end{aligned}$$

Application of a Cartesian-to-spherical coordinate conversion allows $g(x, y, z)$ to be represented as

$$g(u, \theta, \psi) = \Sigma_i A_i \delta(u - u_i, \theta - \theta_i, \psi - \psi_i) \tag{2}$$

of which $g(u)$ is a projection [3]

$$g(u) = \int_{-\pi/2}^{\pi/2} \int_{-\pi}^{\pi} g(u, \theta, \psi) d\theta d\psi \tag{3}$$

where θ is the azimuth angle from point i to the effective radar platform position and ψ is the angle of the effective radar platform position above the x, y plane (ground plane) [3]. Note, the effective radar platform position in the bistatic case is an average of the transmitter and receiver locations. For image formation, the value of u_i is

often defined to be the distance of point i from the origin (scene center) and is called differential range. Differential range is the difference between the range from the antenna to scene center (u_0) and antenna to point i. The differential range is determined by the positions of the transmitter and receiver (not necessarily the same, as in the bistatic case), and the position of point i. In the far-field case, a first-order Taylor expansion can be used to approximate the total differential range as [21]

$$\begin{aligned}\Delta R_i = &- x_i(\cos\theta_{Tx}\cos\psi_{Tx} + \cos\theta_{Rx}\cos\psi_{Rx}) \\ &- y_i(\sin\theta_{Tx}\cos\psi_{Tx} + \sin\theta_{Rx}\cos\psi_{Rx}) \\ &- z_i(\sin\psi_{Tx} + \sin\psi_{Rx})\end{aligned} \tag{4}$$

For each pulse, the received signal $r(t) = s(t) * g(t)$ is match filtered with the transmitted signal $s(t)$ to produce a range profile $g(u) = g(\frac{c\tau}{2\cos(\beta/2)})$ for that pulse. The Fourier transform of the range profile resides in the spatial frequency, or phase history, domain at an azimuth angle θ and grazing angle ψ that average the transmitter and receiver angles where the radar pulse was physically collected. The spatial frequency U is defined as

$$U = \frac{2\cos(\beta/2)}{c}\omega_0 \tag{5}$$

where ω_0 is the center frequency of the transmitted/received signal. Then,

$$G(U) = \mathcal{F}\{g(u)\} = \frac{A}{2}\int_{-u_1}^{u_1} g(u)e^{-j\left\{u\left(\frac{2\cos(\beta/2)}{c}\right)\omega_0\right\}}du. \tag{6}$$

Furthermore,

$$G(X,Y,Z) = \mathcal{F}\{g(x,y,z)\} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g(x,y,z)e^{-j(xX+yY+zZ)}dxdydz \tag{7}$$

$$g(x,y,z) = \mathcal{F}^{-1}\{G(X,Y,Z)\} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} G(x,y,z)e^{j(xX+yY+zZ)}dXdYdZ \tag{8}$$

Often, SAR imaging assumes scatterers lie in the x-y plane and that data is collected in a polar format. Then, the relations

$$\begin{aligned} g(x,y) &= \mathcal{F}^{-1}\{G(X,Y)\} \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} G(X,Y)e^{j(xX+yY)}dXdY \\ &= \int_{-\pi}^{\pi}\int_{0}^{\infty} |U|G(U,\theta)e^{j2\pi U(\Delta R)}dUd\theta \end{aligned} \tag{9}$$

are used to recover the scene reflectivity.

The scene reflectivity $g(x,y)$ may be estimated from the collection of phase history pulses $G(U)$ (or, equivalently, the collection of range profiles g(u)) via Fourier relations or common SAR imaging algorithms which approximate the Fourier relation under data collection limitations, including bandwidth, aperture extent, and sampling.

In practice, frequency is uniformly sampled over a fixed bandwidth, and aperture samples (pulses) are uniformly spaced according to the radar PRF. For circular SAR collections, pulses are uniformly spaced in azimuth angle. Limited bandwidth and aperture extent degrade image resolution, as defined next.

2.1.2 SAR Resolution.

One of the most often looked at criteria for image quality is the resolution of the image, which is measured in both range and cross-range directions. Resolution in the

range direction is defined by

$$\Delta\rho_u = \frac{c}{2B\cos(\beta/2)} \tag{10}$$

where c is the speed of light, B is the bandwidth of the received signal [3, 21]. To define the cross-range resolution, the phase history Cartesian coordinates of bistatic support must be defined first. The coordinates are defined as

$$f_x = f\left(\frac{\cos\theta_{Tx}(\tau)\cos\psi_{Tx}(\tau) + \cos\theta_{Rx}(\tau)\cos\psi_{Rx}(\tau)}{2}\right) \tag{11}$$

$$f_y = f\left(\frac{\sin\theta_{Tx}(\tau)\cos\psi_{Tx}(\tau) + \sin\theta_{Rx}(\tau)\cos\psi_{Rx}(\tau)}{2}\right) \tag{12}$$

$$f_z = f\left(\frac{\sin\psi_{Tx}(\tau) + \sin\psi_{Rx}(\tau)}{2}\right) \tag{13}$$

where f is frequency, τ is the time of each pulse and $\tau = 0$ corresponds to the time of the middle pulse [21]. The bistatic support coordinate system is used to define the effective bistatic aperture [21]

$$\theta_{bi} = \tan^{-1}\left(\frac{f_y(f,\tau_{max})}{f_x(f,\tau_{max})}\right) - \tan^{-1}\left(\frac{f_y(f,\tau_{min})}{f_x(f,\tau_{min})}\right). \tag{14}$$

The bistatic support coordinate system is also used to define the azimuth bistatic look angle $\bar{\theta}_b$ [21]

$$\bar{\theta}_b = \tan^{-1}\left(\frac{f_y(f,0)}{f_x(f,0)}\right) \tag{15}$$

and elevation bistatic look angle

$$\bar{\psi}_b = \tan^{-1}\left(\frac{f_z(f,0)}{\sqrt{f_x(f,0)^2 + f_y(f,0)^2}}\right). \tag{16}$$

The effective aperture and bistatic look angle are used to define the cross-range resolution by

$$\Delta\rho_{cr} = \frac{c}{4f_c \sin\left(\frac{\theta_{bi}}{2}\right) \cos\bar{\psi}_b \cos(\beta/2)} \tag{17}$$

where f_c is the center frequency of the received signal [21].

2.1.3 Sampling Theory.

The physical parameters, such as bandwidth and aperture extent, determine the resolution, but in modern systems the analog physical signals are sampled and digitally represented. Sampling introduces another aspect of imaging that must be accounted for: aliasing. Aliasing is the recurrence of a sampled signal due to the effects of Fourier transforms on band-limited signals. The period of recurrence is determined by the number of samples taken. For the purposes of SAR imaging, this translates to number of samples taken within the bandwidth and the number of pulses (spatial samples) taken along the aperture. The Nyquist criteria state that the minimum sampling rate needs to be twice the highest frequency of interest, otherwise copies of the signal will start to appear within the spectrum of interest [22].

To prevent aliasing in the range direction u of the spatial domain, the frequency domain U sample spacing needs to satisfy [20]

$$\frac{2\pi}{\Delta U} \geq D \tag{18}$$

where ΔU is the sample spacing in cycles per meter of the spatial frequency domain data and D is the diameter of the scene being imaged in meters. Starting from a known or desired D, a maximum ΔU is determined. To achieve the required maximum sample spacing ΔU, the received pulse must be sampled at least M times.

M is found by [20]

$$\Delta U = \frac{8\pi f_{max}}{Mc} \tag{19}$$

where f_{max} is the maximum frequency of the received signal, and c is the speed of light in meters per second.

The total extent in the azimuth direction is the equivalent of bandwidth for the cross-range dimension, and has to be sampled in much the same manner. Each sample in the azimuth direction is a received pulse, so the total number of samples is the total number of pulses that have to be collected. The maximum angular spacing between the samples (for a circular path around the scene) is determined by [20]

$$\Delta\theta \leq \frac{\frac{2\pi}{D}}{\frac{2(2\pi f_{max})}{c}} \tag{20}$$

where $\Delta\theta$ is the sample spacing in the azimuth direction and D is the diameter of the scene [3, 20].

With the data collect and image geometries defined and the parameters that control image quality established, how to process all that information to form an image is now the topic of discussion.

2.2 Image Formation Algorithms

In general, the imaging methods used in SAR can fall into two categories: direct Fourier based algorithms and tomographic based algorithms. The algorithms have both 2-D and 3-D versions for creating images, and in this thesis the 2-D version will be focused on. The 2-D versions assume that the z-direction data is projected onto the (x, y) plane and that the pulse data was collected from a constant grazing angle. All the 2-D imaging methods reconstruct the scene reflectivity $g(x, y)$ from a subset of projections $g(u) = g_{\theta_i}(u)$, where θ_i is the azimuth angle of pulse i and u is the

range, orthogonal to θ_i.

The differences between the direct Fourier methods and the tomographic methods lie in how those projections are processed into an image. A metric that will be used in discussing the algorithms is computational complexity or burden. Computational complexity is a measure of the order of magnitude of operations must be performed to complete the algorithm. In this context, a single "operation" is considered one complex radix-2 FFT butterfly, which consists of a complex multiply and two complex additions. The complex operations equate to four floating point multiplies and six floating point additions [2].

The SAR literature for the direct Fourier techniques is extensive, and there are many direct Fourier techniques available including: the polar format algorithm, the range migration algorithm, the chirp scaling algorithm, and the $\omega - k$ algorithm [2,23]. The direct Fourier imaging methods produce their image by performing a 2-D inverse Fourier transform on the entire collection of phase history data at once. For example, the polar format algorithm first interpolates the collected phase history data from a polar raster to a Cartesian grid, then performs a 2-D inverse Fourier transform to obtain an image $\hat{g}(x, y)$ which approximates the true reflectivity function $g(x, y)$ [2]. The direct Fourier algorithms are a popular choice for SAR image processing due to their low computational burden: on the order of $\mathcal{O}(N^2 logP)$ where N is the dimension of the image (assuming a square grid) and P is the number of pulses gathered [2]. However, the direct Fourier techniques all require the full aperture worth of data to be collected and processed as a whole [7]. To process all the data at once requires a large amount of memory, which limits usage on the constrained platforms being investigated in this thesis. This thesis focuses on the backprojection algorithm, due to the memory and data collection limitations of the direct Fourier techniques.

2.2.1 Tomographic Imaging Methods.

The use of backprojection originated in medical tomographic imaging, such as Computerized Axial Tomography (CAT) scanning. The tomographic technique was adapted for SAR applications in 1983 by David C. Munson Jr., W. Kenneth Jenkins, and James Dennis O'Brien [5]. The work by Munson et al. demonstrated that SAR can be interpreted as a tomographic reconstruction problem, which uses the projection-slice theorem and the inverse Radon transform to form images.

The projection slice theorem states that each demodulated waveform from each look angle "approximates a piece of a one dimensional (1-D) Fourier transform of a central projection of the ground patch at a corresponding projection angle" [5]. Munson et al. refer to computer aided tomography as a two dimensional (2-D) view of a three dimensional (3-D) object through processing many 1-D projectional views from different look angles. While the direct Fourier methods process all the data from all the pulses at each projection angle in a single massive 2-D FFT, the filtered backprojection algorithm performs a 1-D FFT on each pulse separately.

The projection slice theorem operates on the foundational relationship that "the one-dimensional Fourier transform of any projection function $g_\theta(u)$ is equal to the two-dimensional Fourier transform $G(X, Y)$ of the image to be reconstructed, evaluated along a line in the Fourier plane that lies at the same angle θ measured from the X axis" [3]. In SAR, the 2-D match filtered data is collected, processed and stored in a range-angle format (u, θ) and so is treated using a polar coordinate system. As shown in (9), the polar data can be used to directly compute image values g(x,y) using the inverse Radon transform. The representation of $G(X, Y)$ in polar coordinates is [3]

$$G(U\cos\theta, U\sin\theta) = G_p(U, \theta). \tag{21}$$

The Radon transform, in the abstract mathematical sense is the inversion of a function's line integrals to recover the function itself. In application to imaging, the Radon transform "of an image is the mapping of the image into its complete set of projection functions ... while the inverse Radon transform reconstructs the image from its complete set of projections" [3]. The inverse Radon transform in the context of monostatic SAR is defined by [3, 24]

$$g(x,y) = \int_0^{\pi} \left\{ \int_{-\infty}^{\infty} |U| G_p(U,\theta) e^{j2\pi U(x\cos\theta + y\sin\theta)} dU \right\} d\theta. \tag{22}$$

The bistatic case of the inverse Radon transform is

$$g(x,y) = \int_0^{\pi} \left\{ \int_{-\infty}^{\infty} |U| G_p(U,\bar{\theta}_b) e^{j2\pi U(\Delta R)} dU \right\} d\bar{\theta}_b. \tag{23}$$

In SAR, the match filtered received data $g_p(u,\theta)$ is in the spatial domain. Simulated data is often generated directly in the phase history domain as $G_p(U,\bar{\theta}_b)$. The desired end result of the inverse Radon transform is a representation of the spatial domain reflectivity function $g(x,y)$ [3]. The inner integral of (22) is an inverse Fourier transform of the frequency domain data $G_p(U,\theta)$ resulting in the spatial domain projection data $g_p(u,\theta)$. The outer integral of (22) is the backprojection operaton [3] of the spatial domain projection data, which provides the desired spatial domain reflectivity function $g(x,y)$ [3, 20]. Implementation of the inverse Radon transform as described by (22) and (23) is typically referred to as the filtered backprojection algorithm.

From an implementation perspective, the projection slice theorem and inverse Radon transform allow the 2-D FFT to be constructed iteratively on a pulse by pulse basis, which in turn means that only the current pulse being processed and the image itself need to be stored in memory at any given time. The trade off for the reduced memory usage is a significant increase in computational burden. A general breakout

of the computations required for filtered backprojection is as follows. For a data collect of

- P azimuth samples (pulses)
- L frequency bins (samples) for each pulse

the convolution step (Fourier multiplication) requires $\mathcal{O}(PL \log L)$ operations. The data is then used in the backprojection step, which is essentially the summation of the result of the convolution step over each pixel in the image. For an image of

- $N \times N$ pixels
- P projections

the backprojection step requires $\mathcal{O}(PN^2)$ operations. Combining these two steps results in a total number of operations $\mathcal{O}(PL \log L + PN^2)$. If N is significantly larger than L, N^2 is the dominant term, and the order of required operations can be approximated by $\mathcal{O}(PN^2)$ [24]. In much of the literature, it is assumed $P = N$, making the computational burden $\approx \mathcal{O}(N^3)$. This is a significant increase in number of operations (for large P) as compared to the direct Fourier techniques which have a computational complexity of $\mathcal{O}(\log(P)N^2)$.

2.2.2 Factorized Back Projection.

With the computational burden of the filtered backprojection algorithm, its practical use for SAR imaging is limited to cases where fast image creation isn't required and compute power is plentiful. For many applications where time isn't critical, filtered backprojection is preferred over the direct Fourier methods since the Fourier methods assume a small aperture support in phase history domain. As the aperture becomes larger, the polar-to-Cartesian interpolation becomes less accurate. It is possible to process on sub-apertures with Fourier techniques, but that requires more

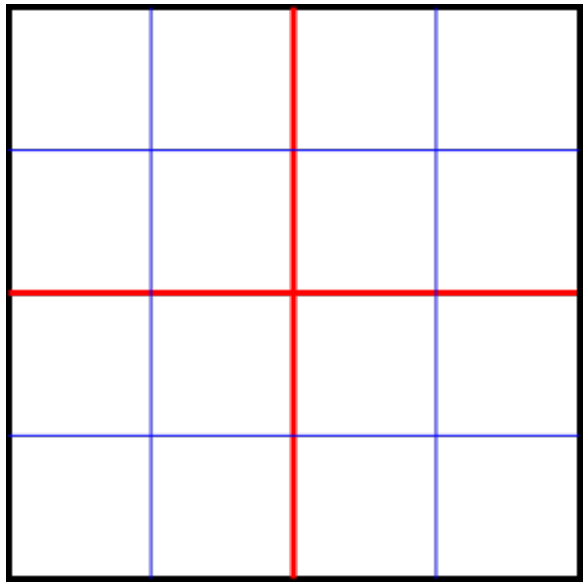


Figure 3. Image partitioning example using two levels of recursion. Red lines indicate quadrant partitions in the first level of recursion; blue lines depict quadrant partitions in the second level.

computations to interpolate and combine the images onto a common grid. To expand the usability of filtered backprojection, a number of fast backprojection algorithms have been developed [4, 7–12, 15, 25].

The basic concept of the fast backprojection algorithms is that for a given image of size N pixels by N pixels, using P collected pulses along the synthetic aperture, the full $N \times N$ image can be constructed by stitching together a number of smaller images created using the same P pulses. For example, if the smaller images are $\frac{N}{2} \times \frac{N}{2}$ then there will be 4 sub-images that can be stitched together to form the whole $N \times N$ image [10, 24]. Figure 3 shows the full image being partitioned into four pieces in the first level of recursion (red dividing lines), and then each of those pieces also being broken into four pieces in the second level of recursion (blue dividing lines).

Subdividing the image doesn't reduce the computational complexity of backprojection, but it is an important first step. It is shown in [10] that subdividing the image recursively and backprojecting onto the smallest image, then combining all the subimages into a full size larger image is equivalent to backprojecting directly onto the

full size image. The computational savings comes from using the angular bandlimit property of the sinogram [9], which is the computerized tomography equivalent of the inverse Fourier transformed phase history data (range profile data) used in SAR. The angular bandlimit property states that:

If $g(u,\theta)$ is supported in $|u| < D$ and $G(U,\theta)$ essentially vanishes for $|U| > f_{max}$, then $F(U,\theta)$ essentially vanishes for $|\theta| > [Df_{max}] + 1$ [9].

Interpreted for the situation at hand, the angular bandlimit property implies that "for a subimage of half-size... the Nyquist sampling rate in θ is also halved" [9]. From a SAR perspective, each of the subimages can be formed using a reduced set of the collected pulses. For the example above, each of the 4 sub-images can be created using $\frac{P}{2}$ pulses, which reduces the overall computational burden by a factor of 2. Applying the image partitioning and pulse reduction recursively is how the factorized backprojection algorithm is able to reduce the computational burden from $\mathcal{O}(PN^2)$ to a theoretical minimum of $\mathcal{O}(\log(P)N^2)$ operations. For each level of recursion, the number of pulses being backprojected onto each pixel is reduced by a factor of two using the angular bandlimit property.

The general procedure for backprojecting a set of P projections onto an image grid is [9]

1. Shift the projection data (range profile) to recenter the data for each of the subimages.

2. Angularly filter and decimate the projection data to yield four sets of $\frac{P}{2}$ projections.

3. Backproject the reduced number of filtered projections to obtain subimages.

4. Combine all subimages to form the final image.

To fit the far-field SAR scenario, a few modifications to the general procedure are required. In the SAR scenario, the data collected are all frequency-offset Fourier data centered at frequency $\frac{2\omega_0}{c}$, giving a bandpass signal in the frequency domain. Another major deviation from general backprojection is that the SAR data are all one-sided Fourier data, so all operations are complex instead of real [9].

To shift the spatial domain data to a new subimage center, the spatial frequency domain phase history data must be multiplied by a phase term. The shifting function is

$$S = e^{j2\pi \frac{f_c}{c}\sqrt{x_0^2+y_0^2}} \tag{24}$$

where f_c is the center frequency of the received signal, and (x_0, y_0) is the center of the subimage being processed. The shifting function S is applied to every sample in the collected phase history data (all azimuth and frequency samples) [10]

$$G'_p(U,\theta) = SG_p(U,\theta). \tag{25}$$

. Once the phase history data has been shifted to the new scene center, the data is filtered and decimated. The filtering is designed to prevent aliasing when reducing the number of samples being used to create the image. An ideal anti-aliasing filter is the sinc function [3]. The filtering operation interpolates new values for the phase history data based on the decimated sampling rate, using

$$G'_{p_m}(U,\theta) = \frac{\Delta\theta}{\Delta\theta'} \sum_{n=-K/2}^{K/2} G'_{p_n}(U,\theta) \frac{\sin(\pi(\theta_m - \theta_n)/\Delta\theta'}{\pi(\theta_m - \theta_n)/\Delta\theta'} \tag{26}$$

where $G'_{p_n}(U,\theta)$ is the value of the nth sample, $\Delta\theta$ is the original sampling interval, $\Delta\theta'$ is the new sampling interval, θ_m are the new sample positions, θ_n are the original sample positions, and K is the total number of samples used in the filter. The filter is

applied as a set of coefficients that are multiplied to the original phase history data in the azimuth dimension. The samples surrounding the new data point being created $G'_{p_m}(U, \theta)$ are multiplied by the corresponding coefficients of the *sinc* function when centered on the new azimuth angle θ_m. Figure 4 visually represents how the filter is applied to the data matrix. After filtering, the decimation step is fairly straight forward: for a decimation of a factor of two, take every other pulse from the filtered phase history data. Once the phase history data has been phase shifted, filtered and decimated, the process is either started over (i.e. shift, filter and decimate again) or if in the last level of recursion, backproject the data onto the subimage grid.

The last level of recursion is determined by a semi-arbitrary limit. The absolute limits are when one of the image dimensions is reduced to one pixel, or when the Fourier data (phase history) is reduced to a single pulse. Any limiting condition between direct backprojection on the whole data set and the absolute limits can be chosen. As more levels of recursion are applied to the process, the computational burden approaches $\mathcal{O}(N^2 \log P)$ [10]. The computational burden is strictly for the backprojection itself. It does not include any of the prepatory steps to decimate and interpolate the phase history data, or to recombine the images. The additional steps shouldn't have a significant computational burden compared to the backprojection step. The caveat of "shouldn't" is stated because the computational burden of the additional steps is dependent on how the steps are implemented.

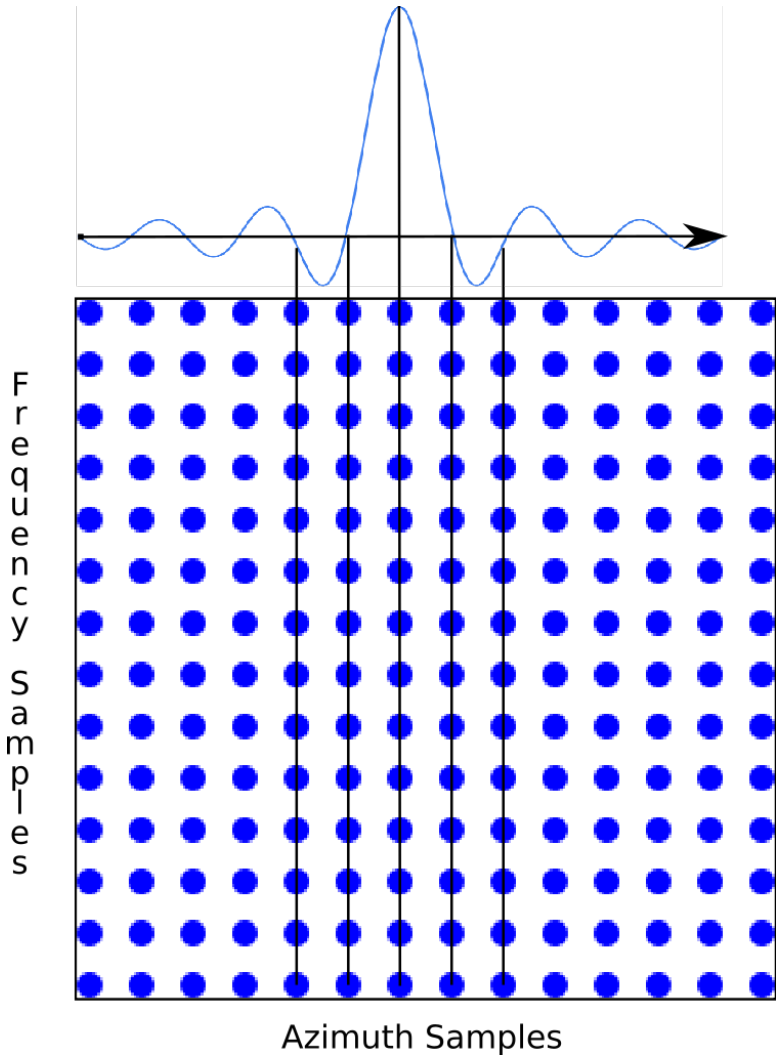


Figure 4. Example of 5-sample anti-aliasing filter applied to phase history data.

2.3 Hardware

Since the 1950's, the Air Force has performed SAR imaging utilizing large airplanes or satellites in conjunction with large computer systems [1,26,27]. These large systems allow for very large scenes to be imaged at very high resolution and in a reasonable amount of time. However, the large systems are expensive to develop and operate, limiting their applications. When the Air Force does attempt to reduce the size of the platform, custom hardware is typically used [19], which is complex and expensive to develop. To reduce cost associated with custom hardware, other avenues

must be explored to find a low cost, low SWAP platform for image processing. In recent years, a number of factors have come together to create an abundance of commercially available platforms that meet the desired criteria. Consumer demand for faster and more capable cell phones, industry demand for low SWAP controllers, a strong hobbyist community, and rapid improvements to the size and speed of transistor technology have all led to development of a wide variety of low SWAP computer platforms.

The choice of hardware has a direct impact on what operating systems and programming languages can be used, and so a discussion of what hardware is being used in this thesis is presented here. A sampling of low SWAP computer platforms include:

- The BeagleBoard and its derivitives
- The PandaBoard
- The LinkSprite pcDuino
- Intel Galileo Gen 2
- FXI Technologies Cotton Candy
- Hardkernel Odriod-CI
- Raspberry Pi

Each of these different systems has its pros and cons. Some, like the PandaBoard, have much better processors and more memory. But its enhanced compute power comes at a price, literally: the PandaBoard retails for approximately $200 while at the low end of the cost spectrum, the Raspberry Pi retails for $25. Almost any of these platforms could have been used for this thesis. However, due to budgetary and departmental constraints, the Raspberry Pi Model B (R-Pi) is used for this project.

The R-Pi is a fully functional computer on a board the size of a credit card. The system is based around the Broadcom BRCM2835 System-on-Chip (SOC) which is in turn based around an ARM 1176 processor. The pertinent specifications of the

R-Pi used for this thesis are [28]:

- BRCM2835 SoC at 700MHz
- 512MB RAM
- 2 USB ports
- 17 available GPIO pins
- 700mA, 3.5W power usage
- Physical Dimensions: $85 \times 56 \times 17$mm.

To use the R-Pi, as with any computer, an operating system (OS) must be installed. Thanks to a strong open source community, there are a number of OSs available for the R-Pi. One of the better supported and commonly used OSs is Raspbian, an unofficial port of the Debian operating system with optimizations for running on the hardware of the R-Pi [29]. To create code that would be compatible with the Raspbian OS, initial development of the code is completed on a similar operating system: MacOS 10.6. Creating code for an open source OS gives a significant amount of freedom in the choice of programming language while keeping cost low.

III. Code Development and Methodology

In Chapter 1 the motivation to create a low cost and low size, weight, and power (SWAP) radar platform is given. This chapter explains the choice of hardware, programming language, and the implementation of both the filtered and factorized backprojection algorithms.

3.1 Language Choice

There are many different programming language choices for implementing the backprojection algorithms, each with their pros and cons. While many criteria can be used for supporting a chosen language, this thesis focuses on three parameters in determining which language would be best suited for implementation on the Raspberry Pi. Using parameters of memory usage, speed of execution, and compatibility, the language types to be evaluated are `MATLAB`® , interpreted languages and compiled languages. `MATLAB`® is separated from other interpreted languages because the interpreter is only a small part of the suite of tools that `MATLAB`® has available.

3.1.1 Memory Usage.

The first parameter, memory usage, is determined by the amount of Random Access Memory (RAM) used by the system to create the image data. The RAM used includes any memory used by a parent application. To begin, `MATLAB`® , specifically current versions, utilizes a large amount of RAM to run. `MATLAB`® scripts are typically executed within the user interface; however, the user interface includes additional tools that may not be necessary for the script being executed. In most workstations the memory overhead is not an issue, but in the case of a platform such as the Raspberry Pi any excess memory usage becomes a great hindrance. For instance,

MATLAB® 2014a uses $\approx$ 400MB of RAM upon launch. For a robust workstation (i.e. large scale SAR image processing) that has $8 - 16$GB of RAM, the memory to run MATLAB® is easily acceptable. Unfortunately, the Raspberry Pi only has 512MB of RAM; a single program using 400MB would cripple the platform. Another feature of MATLAB® that could be used is the MATLAB® compiled runtime (MCR). The MCR provides the ability to compile a script into a semi-self-contained executable [30]. This executable uses significantly less RAM because the full user interface is not required to run. However, the executable does require the MATLAB® compiled run-time (MCR) environment to be installed on the system where it is to be run [30].

In contrast to the use of MATLAB® , interpreted languages such as Python may be better suited for a low memory project [31]. The user interface for languages like Python is often a command line prompt, using significantly less RAM than MATLAB® . In addition, scripts can be run without the command prompt in a similar fashion to the MATLAB® compiled executables. Combining the decreased RAM usage with the ability to run scripts in a similar fashion to MATLAB® makes a Python-like language an appealing choice.

Finally, compiled languages such as C and C++ use the least amount of RAM in comparison to the languages discussed above. With compiled languages, the code itself cannot be run like the scripts for the above languages. Instead the code must be processed by a compiler to create an executable file. The executable file created is completely independent, and able to be run on any compatible system. Since the executable does not need any external interface or helper, there is no memory overhead beyond what has been specifically called for in the code. For limited systems like the Raspberry Pi, reducing memory usage is critically important.

3.1.2 Speed of Execution.

The second parameter, speed of execution, is defined in the context of this thesis as the time required to process the phase history data into an image. Execution time is an essential parameter for this thesis, as the future goal of the project would be to generate SAR imagery in real-time. `MATLAB`® , which specializes in vector and linear algebra type operations, is able to generate image data in a relatively short amount of time [30]. The interpreted languages, as stated above, are converted to machine-specific instructions at runtime, which adds to the execution time of the script/code [31]. The compiled languages have to be converted to machine-specific instructions before execution, resulting in faster runtimes than the interpreted languages. When compiled without third party linear algebra libraries, the compiled languages and `MATLAB`® run in approximately the same amount of time, though the difference in run times is close for all three language options.

3.1.3 Compatibility.

The third parameter, compatibility, has a much broader scope. In this context, compatibility will be broken out into two areas: internal and external compatibility. Internal compatibility allows script/code to continue working between different versions of the language it is written for. External compatibility is the ability of the script/code to function on different platforms.

3.1.3.1 Internal Compatibility.

All three language families being evaluated lack consistent internal compatibility. In the case of `MATLAB`® , commands and functions used in the scripts may or may not be compatible with different versions of `MATLAB`® . Some commands and functions are deprecated or completely removed, while others are altered to the point they no

longer behave the same as previous versions. Similarly, if the MATLAB® scripts are compiled to run with the MCR, then the compiled scripts must be executed on a system with the compatible version MCR [30].

For interpreted languages, specifically Python, compatibility is better. For Python the major releases are internally compatible, but Python is not compatible between major releases [31]. For example, Python version 2.3 is compatible with version 2.6 but not compatible with version 3.2. Currently, the two major releases for Python are 2.x and 3.x, both with numerous minor releases so care must be taken to ensure that the correct major release of Python is installed [31].

Compiled languages function a bit differently than interpreted languages. General code written for an old version of the language will compile in the new version, and in some cases different languages can use the same compiler. However, it is not always the case that code written for the new version will be able to compile with an older compiler.

3.1.3.2 External Compatibility.

For external compatibility, the specific hardware is not an issue for any of the choices. However, this does not mean that scripts and programs can be used in a cavalier fashion. MATLAB® is a licensed product, so each installed instance of it needs to be licensed to operate [30]. Depending on the method of licensing, there may be a limit on the maximum number of subscriptions an organization can use, or additional fees for each new installation. Utilizing a license on a simplified platform such as the R-Pi is impractical at best considering the crippling amount of memory MATLAB® will use. Python does not have the constraint of licensing, as it is a completely open source tool [31]. All that is required to run Python code is for the Python interpreter to be installed, and many operating systems come with Python as a preinstalled package.

Compiled languages are a little different; because code for a compiled language can only be executed after being compiled and saved as machine-specific instructions, the code must be recompiled for the different types of hardware and operating systems that it will be run on.

Given the hardware platform being used in this thesis, memory usage and speed to execute are by far the most important factors. The R-Pi has a limited amount of both RAM and processing power [28]. Running the full `MATLAB`® program on the R-Pi would use the vast majority of the available RAM, debilitating the R-Pi, so the full `MATLAB`® is not a viable option. Using the `MATLAB`® compiled runtime is a potential option, as it requires much less memory to run. However, the proprietary nature and its compatibility challenges make this choice less appealing. The interpreted languages provide another good option, but the minor overhead of the interpreter hinders the performance. This leaves the family of compiled languages. Any of the available compiled languages would be suitable for this project, but due to familiarity and experience at this time, C++ will be the language used to implement the backprojection algorithms.

3.2 Filtered Backprojection Algorithm Development

The filtered backprojection algorithm is a well-known and established image processing algorithm. The implementation currently used at AFIT is a `MATLAB`® script developed by Dr. Julie Jackson, a major extension to a previous `MATLAB`® script developed by LeRoy Gorham and Linda Moore from Air Force Research Laboratory Sensors Directorate [32]. Dr. Jackson's backprojection algorithm provides a more generalized and mathematically rigorous implementation of the prior script, to include [33]:

- Properly zero padding the bandpass frequency domain data for use in the inverse

FFT function [34]

- Bistatic SAR (with monostatic as a limiting case)
- The appropriate 2-D or 3-D ramp filtering required by the inverse Radon transform [3] to account for integration sample spacing in phase history.

Dr. Jackson's work creates the foundations from which this project is built. The research that occurs in this thesis will be focused around three stages of code adaptation and testing. Stage one will take Dr. Jackson's `MATLAB`® script and translate it to C++ code that can be supported by the current project hardware (the R-Pi). After adapting and compiling, the C++ version will be compared against the `MATLAB`® version for speed of execution and image quality. If the backprojection algorithm is not correctly translated, then the C++ code will not aid in the development of a low cost, low SWAP SAR platform. On the other hand, if the C++ code does function at a similar speed, with acceptable image quality, it can then be used as a baseline for further development of the desired platform.

For translation of Dr. Jackson's `MATLAB`® script into the C++ programming language, the major flow of the algorithm is the same. Figure 5 shows the flow for the C++ implementation of the filtered backprojection algorithm. The "backprojection" block of Figure 5 is a subroutine within the algorithm that performs the backprojection operation as outlined in Chapter 2, the steps of this subroutine are detailed in Figure 6. The shaded section within the backprojection subroutine is the loop that adds each processed pulse to the image (i.e. the actual backprojection operation). Within the backprojection subroutine there is an additional process that performs the inverse FFT operation, also detailed in Figure 6.

The primary deviation from the `MATLAB`® script is in the creation of phase history data. The C++ algorithm reads in the data from a file while the `MATLAB`® algorithm

generates the phase history within the script. However many changes are made in how the individual variables are handled. Without the use of third party linear algebra libraries, and without having the MCR available, all of the vectorized computations that are inherent in `MATLAB`® have to be broken out into loops that only operate on one element of the vector at a time. The vector to loop translation must be applied to many of the variables in the algorithm, including the creation of the receiver frequency and azimuth vectors and the transmitter azimuth vector. It is assumed that the frequency band of the receiver and transmitter match. Other operations that require loops or nested loops are: the creation and application of the ramp filter; any of the shifting functions; and for each pulse looping through each pixel of the image determining which index of the range profile data matches the range from scene center of the pixel.

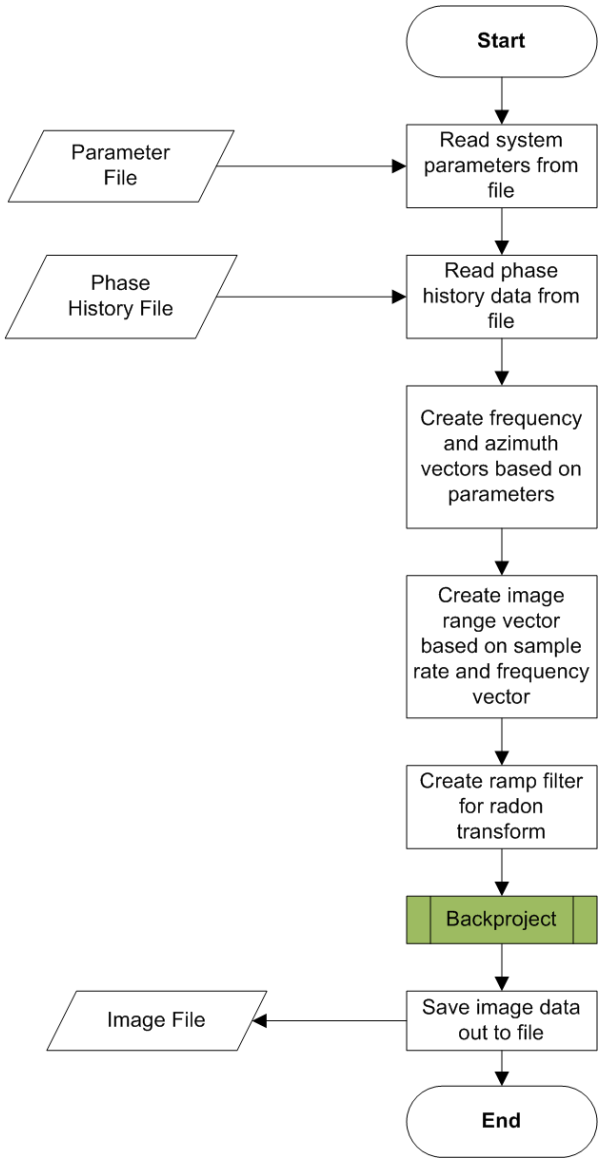


Figure 5. Flow chart for main program block of filtered backprojection. The backprojection subroutine is depicted in Figure 6.

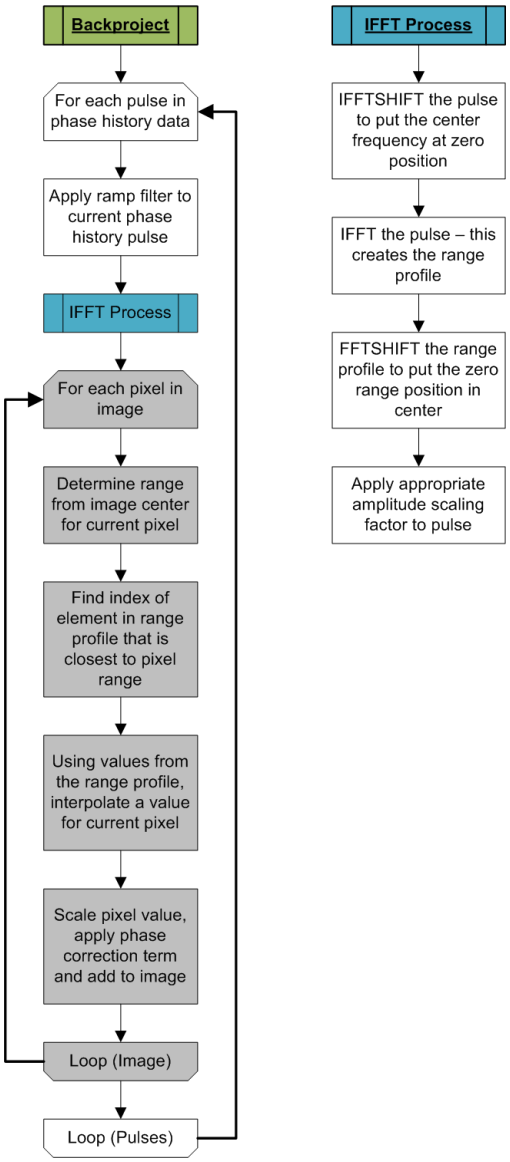


Figure 6. Flow chart for sub-routines of filtered backprojection, i.e. FFT procedures and pixel summation.

3.3 Performance of Filtered Backprojection Implemented in C++ vs. MATLAB®

The filtered backprojection algorithm is implemented in C++ in order to establish a baseline version of code that most directly compares to the MATLAB® script developed by Dr. Jackson [33]. As described above, modifications to the MATLAB® script had to be made to properly translate the filtered back projection algorithm over to the C++ language. The performance of the C++ implementation is measured both through comparison of the image data generated and the time required for the C++ implementation to generate the image data.

For the simulated image creation, a control set of ideal phase history data is generated using a bistatic system. Figure 7 shows the locations of the scatterers in the scene being imaged. All the targets have unit amplitude. Four targets in the lower left quadrant are off set from the (x, y) plane to show handling of layover effects. The targets at $(x, y) = (2.5, -2.5)$ and $(2.5, -7.5)$ are located at $z = 3$ above the plane. The targets at $(x, y) = (7.5, -2.5)$ and $(7.5, -7.5)$ are located at $z = -3$ below the plane.

The signal used has a center frequency of 10 GHz, with a bandwidth of 2 GHz which gives a range resolution of $\approx$ 0.075 m for the monostatic case according to Equation (10) from Chapter 2. The bistatic case has a range resolution of $\approx$ 0.1 m according to Equation (10) using the bistatic angle. To prevent aliasing in the range direction for an image of +/-10 m (D = 20 m diameter), Equation (18) gives the minimum sampling spacing as 0.314 cycles per meter. Using Equation (19) and solving for M, the minimum number of samples for the 20 m diameter image is 533 samples so using 1024 frequency samples (the next power of two) is more than satisfactory [3, 20].

For the cross-range or azimuth direction, the synthetic aperture for the monos-

tatic case has an angular extent of 30°, or 0.5236 radians, which gives a cross-range resolution of ≈ 0.03 m according to Equation (17) from Chapter 2. The bistatic case has an angulare extent of 15° or 0.2862 radians, which gives a cross-range resolution of ≈ 0.06 m. To prevent aliasing in the cross-range direction, the maximum sample spacing along the data collection track/aperture is 6.82×10^{-4} radians, from Equation (20) from Chapter 2. The simulation data has 2048 evenly-spaced azimuth samples (pulses) giving a sample spacing of $\Delta\theta = 2.56 \times 10^{-4}$ radians, which is less than the maximum sample spacing stated above. A quick lookup of the above parameters can be found in Table 2. A bistatic case using the same set of targets from Figure 7 is also run using the parameters in Table 3.

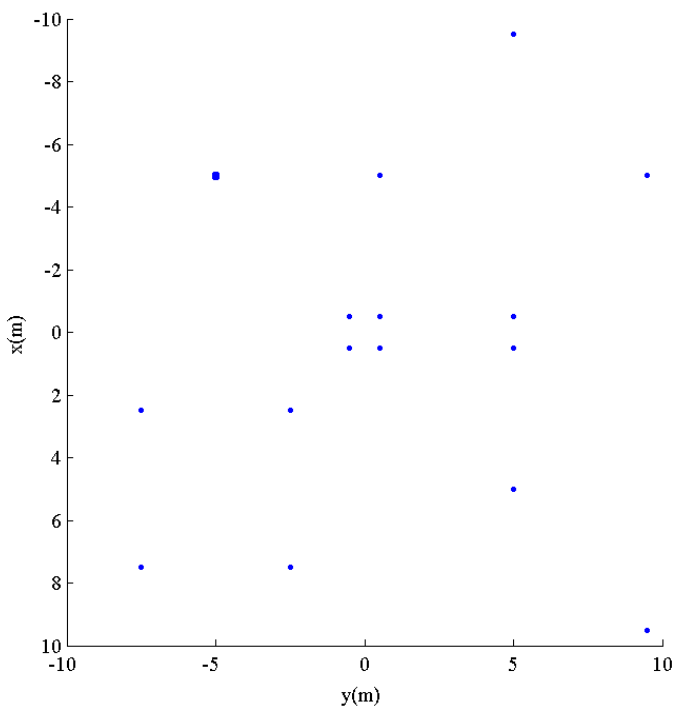


Figure 7. Location of targets in simulation scene. All the targets have unit amplitude. Targets at $(x, y) = (2.5, -2.5)$, $(2.5, -7.5)$, $(7.5, -2.5)$ and $(7.5, -7.5)$ are located off the (x, y) plane and appear at projected locations in the formed image.

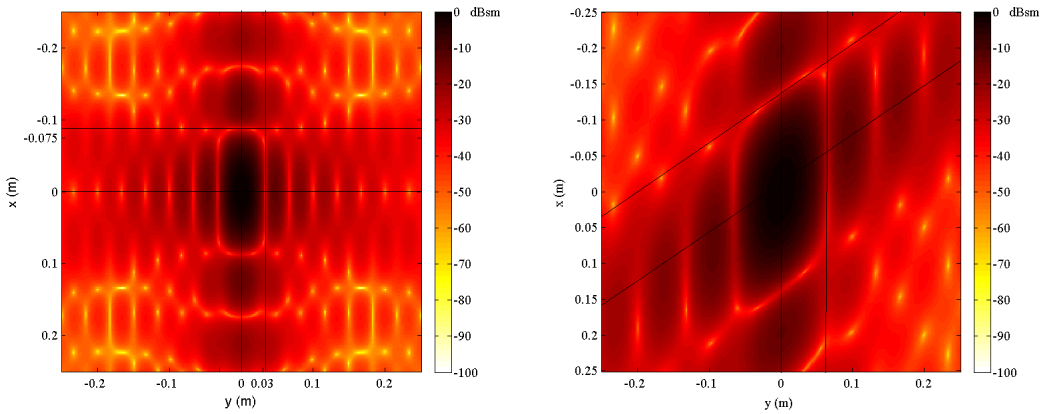
Table 2. Table of Parameters for Monostatic Simulation

Center Frequency	10 GHz
Bandwidth	2 GHz
Bandwidth Samples	1024
Tx Azimuth Extent	30°
Tx Center Azimuth	0°
Rx Azimuth Extent	30°
Rx Center Azimuth	0°
# Pulses	2048
Tx Elevation	30°
Rx Elevation	30°
Image Extent	20 m ×20 m
# Image X pixels	256
# Image Y pixels	256

Table 3. Table of Parameters for Bistatic Simulation

Center Frequency	10 GHz
Bandwidth	2 GHz
Bandwidth Samples	1024
Tx Azimuth Extent	0°
Tx Center Azimuth	75°
Rx Azimuth Extent	30°
Rx Center Azimuth	0°
# Pulses	2048
Tx Elevation	30°
Rx Elevation	30°
Image Extent	20 m ×20 m
# Image X pixels	256
# Image Y pixels	256

For a verification that the resolution of a point target in the monostatic case matches what the parameters in Table 2 dictate, a zoomed in image of a single point target is generated and the distance from the center of the point target to the edge of the first contour is measured, as shown in Figure 8(a). The cross range direction (labeled y-axis) has a point spread of $\approx$ 0.03m and the range direction (labeled x-axis) has a point spread of $\approx$ 0.0791m, which matches what is expected from the parameters. For a verification that the resolution of a point target in the bistatic case



(a) Monostatic Resolution Cell

(b) Bistatic Resolution Cell

Figure 8. Zoom in of point scatterer to show resolution cell size.

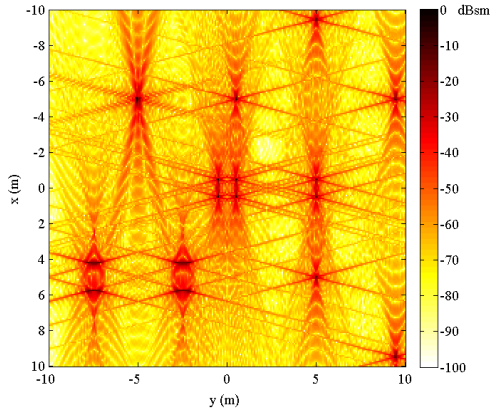
matches what the parameters in Table 3 dictate, a zoomed-in image of a single point target is generated and the distance from the center of the point target to the edge of the first contour is measured, as shown in Figure 8(b). The cross range direction has a point spread of $\approx$ 0.06m and the range direction (offset bacause of the bistatic look angle) has a point spread of $\approx$ 0.1m, which matches what is expected from the parameters.

Using the parameters outlined in Table 2, images are created for both the `MATLAB`® script and the C++ code so that a direct comparison can be made between the two implementations. The image for the `MATLAB`® script can be seen in Figure 9(a), and

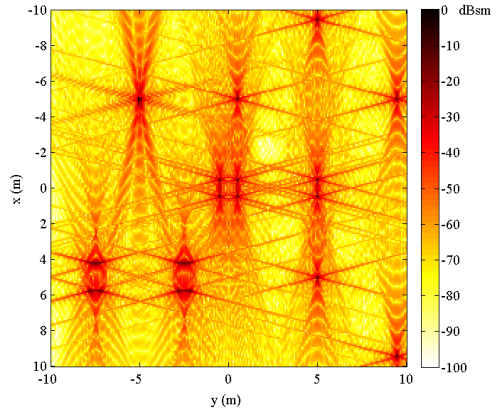
the image for the C++ code can be seen in Figure 9(b). The processed pulse data (range profile data) is complex when added to each pixel value, which doesn't plot into an image well. The magnitude of the pixel values on a decibel (dB) scale is used for visualization. Visually there appears to be no difference between the MATLAB® generated image data and the C++ generated image data, but a direct comparison of the pixel values tells a slightly different story. Figure 9(c) shows the magnitude of the complex difference between the MATLAB® generated image and the C++ generated image in dB. The maximum linear value of the difference is $-0.00026428-0.00044034i$ or an absolute value of 5.1356×10^{-4} (-66dB), and its mean-squared-error (MSE) is $1.7578 \times 10^{-12} + 6.6643 \times 10^{-12}i$ or an absolute value of 6.8922×10^{-12} (-223dB).

For a bistatic case using parameters from Table 3, the maximum difference between the MATLAB® and C++ images is $0.0045 + 0.001i$ (-47dB) and a MSE of $-1.8989 \times 10^{-10} + 8.8717 \times 10^{-11}i$ (-194dB). Figures 10(a) and 10(b) show the MATLAB® and C++ generated images for the bistatic case. Figure 10(c) shows the difference between the MATLAB® and C++ images for the bistatic case.

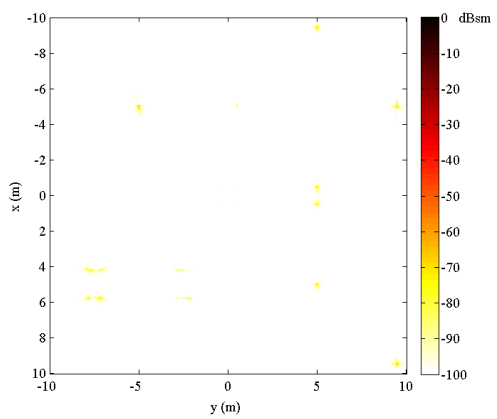
A potential cause for the difference between the MATLAB® and C++ generated images is the differences in how MATLAB® and C++ have their constants defined. For example, the C/C++ math library and MATLAB® define π the same out to 15 decimal places, but then their values for the definition of π differ, which is an error that can propagate through calculations using π.



(a) MATLAB® generated image data

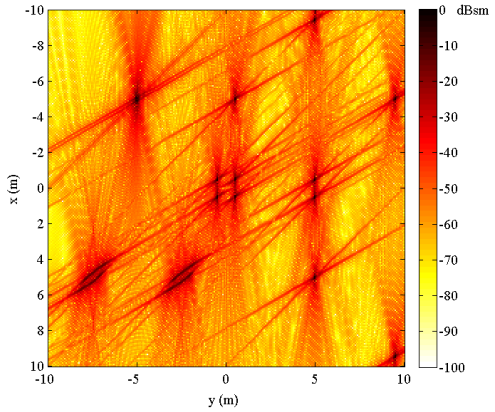


(b) C++ generated image data

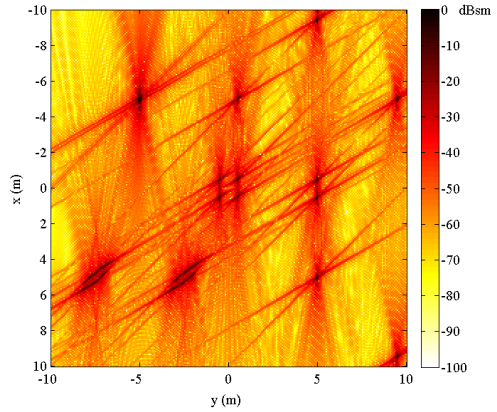


(c) Magnitude of complex difference between MATLAB® generated image in 9(a) and C++ generated image in 9(b)

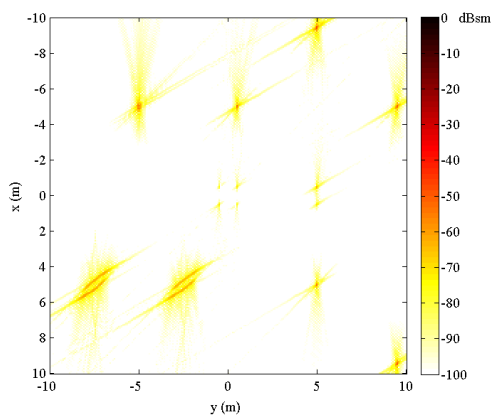
Figure 9. Comparison of MATLAB® generated image and C++ generated image for the monostatic case.



(a) MATLAB® generated image data



(b) C++ generated image data



(c) Magnitude of complex difference between MATLAB® generated image in 10(a) and C++ generated image in 10(b)

Figure 10. Comparison of MATLAB® generated image and C++ generated image for the bistatic case.

With the low margin of error established, it is assumed that a reasonable baseline version of C++ code has been achieved. Under this assumption, the time to generate image data between the `MATLAB`® script and the C++ program is compared to establish the performance of unoptimized C++ code with no linear algebra libraries. Generating the images on the code development laptop, the `MATLAB`® script has an average time to run of 4 min 34 sec over 100 runs, and the C++ code has an average time to run of 4 min 55 sec over 100 runs. The difference in the run times demonstrates some of the optimizations `MATLAB`® has for dealing with vectorized data. Unfortunately, as stated earlier in this chapter the extra memory needed for `MATLAB`® to run prohibits the use of `MATLAB`® on a platform such as the Raspberry Pi.

From a system resources perspective, memory and processor usage are the important metrics to pay attention to. For the C++ code, the majority of the memory being used is determined by the size of the phase history data and the image. Each data point is represented by a complex double variable, which uses 16 bytes of memory — an 8 byte double for the real component and an 8 byte double for the imaginary component. For the image, the memory used is $N_x \times N_y \times 16$ bytes where N_x and N_y are the pixel dimensions of the image. For the phase history data, the memory is based on the dimensions of the phase history array (number of azimuth positions $\times$ frequency samples). For the test simulation above, the memory usage for just the image and phase history data sets is: $256 \times 256 \times 16$ bytes for the image and $2048 \times 1024 \times 16$ bytes for the phase history data. Using standard power of two notation common in the computing world, the image uses 2^{20} bytes or 1 megabyte (MB) and the phase history data uses 2^{25} bytes or 32 MB. In total, the approximate minimum memory usage of the filtered backprojection C++ code, as currently implemented for the example in 9(b), should be $\approx$ 35 MB. A quick review of resource usage between `MATLAB`® and C++ can be seen in Table 4. It should be noted that

the monostatic and bistatic cases use the same image size, same number of frequency samples and same number of azimuth positions; therefore the amount of data being used in the algorithm is the same. The time and memory usage between the mono and bistatic cases is equivalent.

Table 4. C++ resource usage versus MATLAB® .

Resource \Language	MATLAB®	C++
Time	4 min 34 sec	4 min 55 sec
Memory	$\approx$ 600 MB	32.88 MB

The actual run-time memory usage of the C++ implementation for the example in 9(b) is 32.88 MB, using the simulation data sets. For the MATLAB® script in [33], the memory usage of just MATLAB® R2014a is $\approx$ 450 MB and with the code running the memory increases to $\approx$ 600 MB, which indicates that the MATLAB® script uses $\approx$ 150 MB of memory. Alternative implementations may be able to decrease the amount of memory used. However, these alternatives could significantly increase the total processing time. Due to both memory and time being pertinent criteria for the project, the current tested model will continue to be used. Alternative implementations will be discussed later as possible future research options.

3.4 Factorized Backprojection Algorithm Development

The factorized backprojection (FBP) algorithm as described in Chapter 2.2.2 modifies the filtered backprojection algorithm to decrease the amount of computation required to form the desired image. To implement the FBP algorithm, a few critical changes had to be made to the algorithm for filtered backprojection. The prime motivation for using the FBP algorithm is to decrease the total time required to generate an image. As detailed in Chapter 2, FBP decreases computation time through recursive decimation of the angular samples and sub-image processing. A flow diagram

of the recursive process is referenced in Figure 11. In the "factorize" subroutine, the shaded blocks represent the processing of each image quadrant.

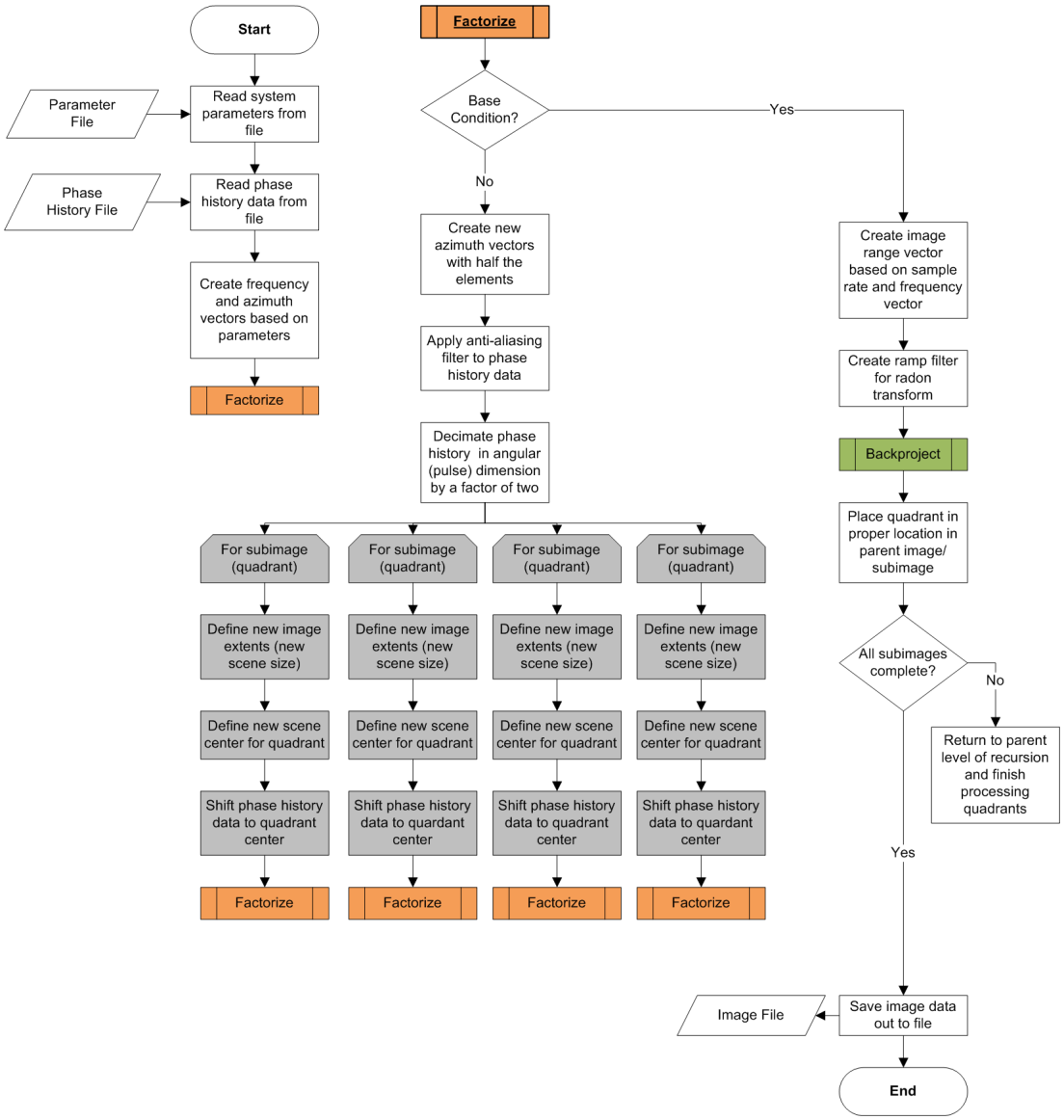


Figure 11. Flow chart for main program block of factorized backprojection.

The major change translating from the filtered backprojection to the factorized backprojection is the implementation of the recursive decimation and partitioning steps. The first check within the recursive function is to check to see if the base condition has been met. The base condition(s) can be: a fixed number of recursion levels, a minimum pixel number for the sub-image passed into the function, or a

minimum number of pulses to process. The recursive function as implemented checks for all three conditions. If the base condition(s) are met, then backprojection is performed on the current sub-image and the decimated phase history data; otherwise the recursive function is entered again.

The phase history decimation occurs over the azimuth samples (pulses) and reduces the number of samples by a factor of two. Sampling theory, as detailed in Chapter 2, tells us that the number of samples within the bandwidth determine the un-aliased scene extent. To mitigate the effects of aliasing, sampling theory recommends implementing an anti-aliasing filter, with an ideal filter being an infinite length sinc function [3]. For practical cases, the sinc function must be truncated. Where the filter is truncated at is a compromise between speed and aliasing. The FBP code implements various length filters to highlight the effect of aliasing and filter length on image quality, and will be discussed further in the next section.

Once the phase history data has been filtered and decimated for the current level of recursion, the image at the current level of recursion is divided up into four quadrants by limiting each quadrant to half the current image scene extent and half the pixel count. Each quadrant is iterated through to process the image data for that quadrant, as was shown in Chapter 2.

3.5 Performance of Factorized Backprojection in C++ vs. Filtered Backprojection in C++

Performance of the C++ factorized backprojection implementation will be evaluated using the same criteria as the C++ filtered backprojection: speed, image quality and memory usage. As stated in the previous section, the purpose of the factorized backprojection algorithm is to reduce total computation time. However, the method used to reduce the computation time introduces error in the form of image aliasing.

This prior knowledge leads us to focus on the computation time and image quality instead of memory, which was the focus of the stage one above.

To evaluate the computation time, a minimal length anti-aliasing filter (3 samples) is used and the number of recursion levels is varied. As stated in Chapter 2, the number of recursion levels should reduce the computation time, up to a point. Figure 12 shows the time to compute for multiple levels of recursion. In the figure a minimum time to compute is seen at two levels of recursion, but the general curve shown indicates a minimum somewhere between 2 and 3, matching the minimum of $e \approx 2.718$ stated in [7]. The theoretical minimum cannot be directly tested for in the current implementation as a non-integer number of recursions cannot be done. Above three levels of recursion the number of branches generated in the recursive tree becomes the largest time consumer and the computation time increases. Each branch represents one quadrant of the image/subimage that is being processed for the current level of recursion. As the number of recursions increases, the number of branches increases at a rate of $4^R + 1$ where R is the number of recursions. Backprojection of the phase history data ends up being fewer processing steps than the over head processing to; filter and decimate the phase history data for each level of recursion, create new subimages, and pass the required variables to the next level of recursion.

Now that the factorized backprojection algorithm has been shown to decrease the computation time, what penalty is paid in image quality to achieve that time decrease? As discussed in Chapter 2, decreasing the number of samples reduces the un-aliased scene extent and requires an anti-aliasing filter. For a minimal length filter of 3 samples, more levels of recursion increase the aliasing error rapidly. Figures 13(a), 13(b), 13(c), and 13(d) visually depict the how the image quality suffers from the aliasing. Figure 15 shows how the mean-squared error (MSE) from aliasing grows with recursion levels. The MSE is computed the same as in previous image comparisons:

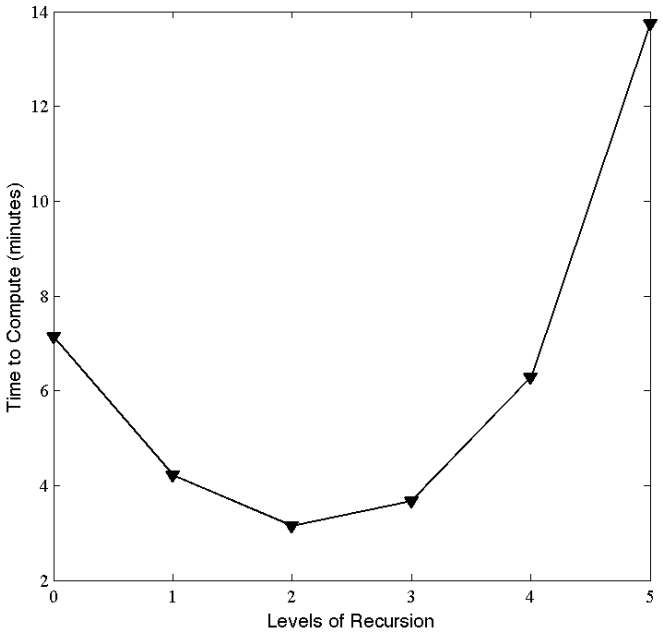
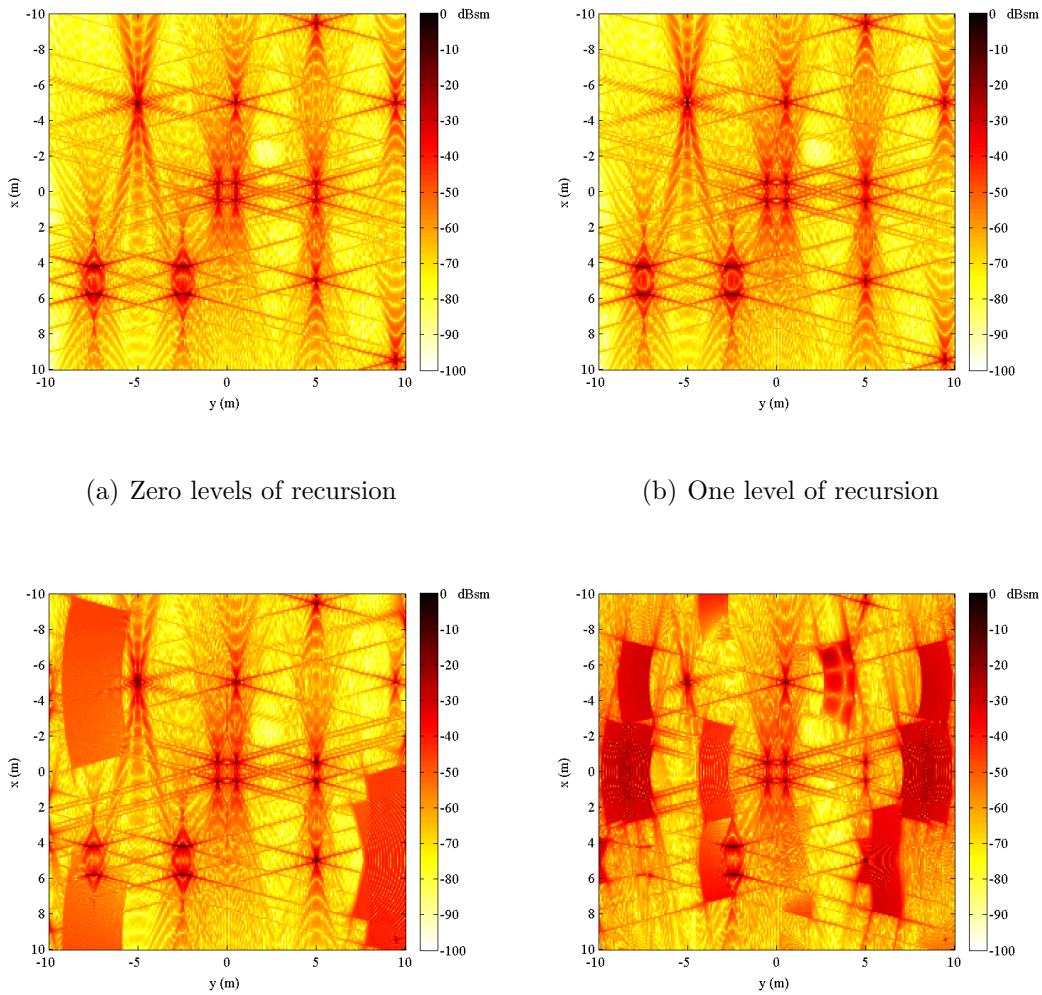


Figure 12. Time to compute vs. number of recursion levels for the example in Figure 9(b).

using the difference of the complex magnitude of each corresponding pixel. The error introduced by the pulse reduction is significantly larger than the error due to converting from `MATLAB`® to C++ (-81.6dB versus -223dB after just the first level of recursion). Zero levels of recursion is equivalent to filtered backprojection, and has the same error as in Figure 9(c)



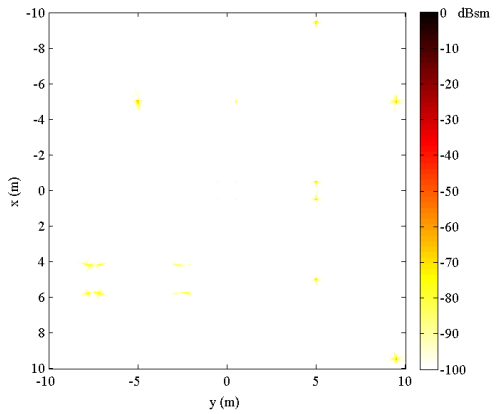
(a) Zero levels of recursion

(b) One level of recursion

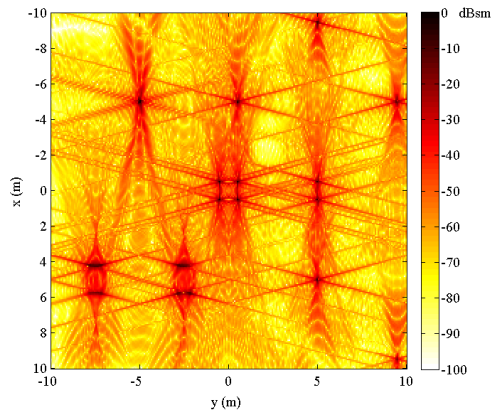
(c) Two levels of recursion

(d) Three levels of recursion

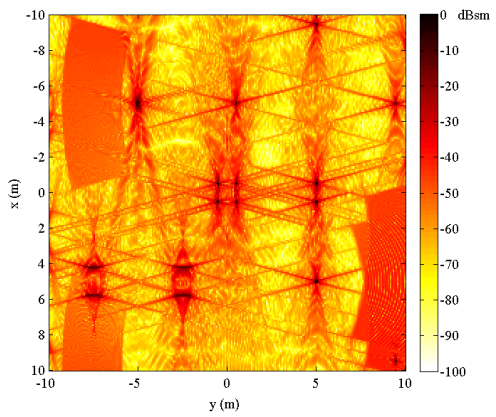
Figure 13. Effect of aliasing on image quality for various levels of recursion.



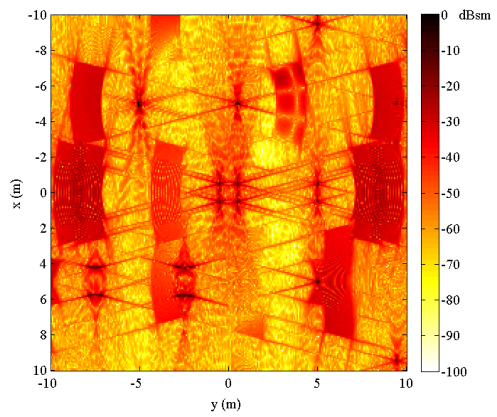
(a) Zero levels of recursion — complex difference from MATLAB® filtered backprojection



(b) One level of recursion — complex difference from MATLAB® filtered backprojection



(c) Two levels of recursion — complex difference from MATLAB® filtered backprojection



(d) Three levels of recursion — complex difference from MATLAB® filtered backprojection

Figure 14. Aliasing error introduced through various levels of recursion. Zero levels of recursion is equivalent to filtered backprojection (no aliasing), and has the same error as in Figure 9(c).

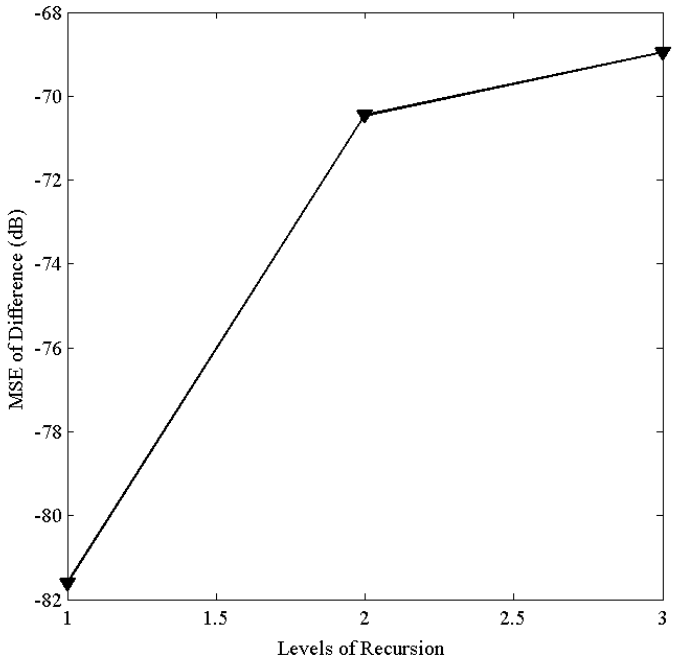


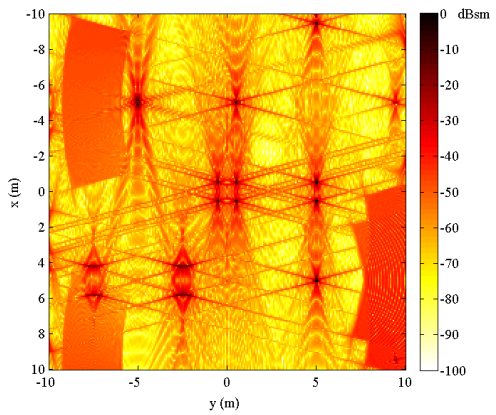
Figure 15. MSE of the aliasing error introduced by recursion.

The rapidly growing error in Figure 15 succinctly shows that something must be done about the aliasing issue, supporting sampling theory's call for an anti-aliasing filter. This filter can be implemented in a number a ways, with the ideal case being the infinite length sinc filter described in Chapter 2. Since an infinite length filter cannot be implemented in code, a truncated sinc function is used as an approximation for the ideal case. Where to truncate the sinc function is an arbitrary decision, and is influenced by code complexity, computation time and desired image quality. Chapter 2 posits that a longer filter will reduce aliasing more. However, a longer filter is more complex to implement in code, and increases total computation time. It is desirable to see the impact of the anti-aliasing filter to determine the benefits of implementing a longer filter.

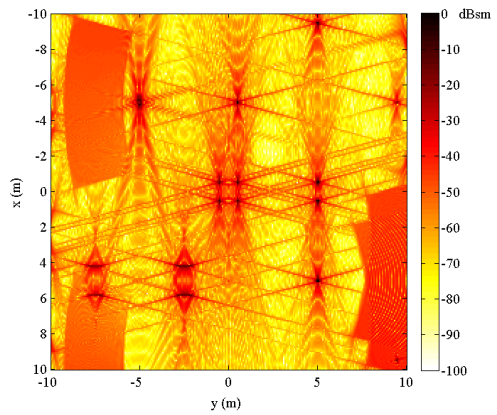
To see the impact that the anti-aliasing filter has on the image quality, filters of various lengths are implemented. To show aliasing in the scene, but also keep the

computation time to a minimum, the number of recursions is held to 2. The phase history data is a discrete, sampled data set, so a discrete anti-aliasing filter must be used. To apply a discrete filter, a set of filter coefficients to match the length of the filter are determined using Equation (26) in Chapter 2 with a new sampling interval. The sampling interval used here is the desired angular distance between every other pulse in radians (the new sampling rate). The distance from the peak (center) of the sinc function to first zero crossing point is determined by the new sampling interval, and detailed in Chapter 2. The azimuth sample data is multiplied by the value of the sinc function at the corresponding location. Figures 16(a), 16(b), and 16(c) show the result of filters with $n = 3, 5, 7$ samples respectively. Note that as the length of the filter is increased the aliasing artifacts do not disappear, but they are reduced. As the length of the filter is increased, the aliasing error will be reduced. There are also other filtering and windowing functions that can be implemented to recude the error, as discussed in [17].

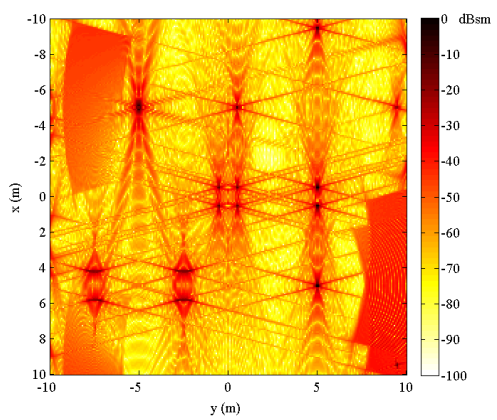
Setting the number of recursion levels at two, the anti-aliasing filter length is altered to see the effect on the time to compute and the image quality. The longer the filter used, the better the image quality but the longer the run-time of the algorithm. The effect of the length of the anti-aliasing filter on image quality can be seen in figure 17, which shows that as the filter length increases, the error decreases. The time to compute wasn't greatly affected for the lengths of filter chosen (on the order of milliseconds).



(a) Length 3 anti-aliasing filter



(b) Length 5 anti-aliasing filter



(c) Length 7 anti-aliasing filter

Figure 16. Images generated using 2 levels of recursion and varied filter length.

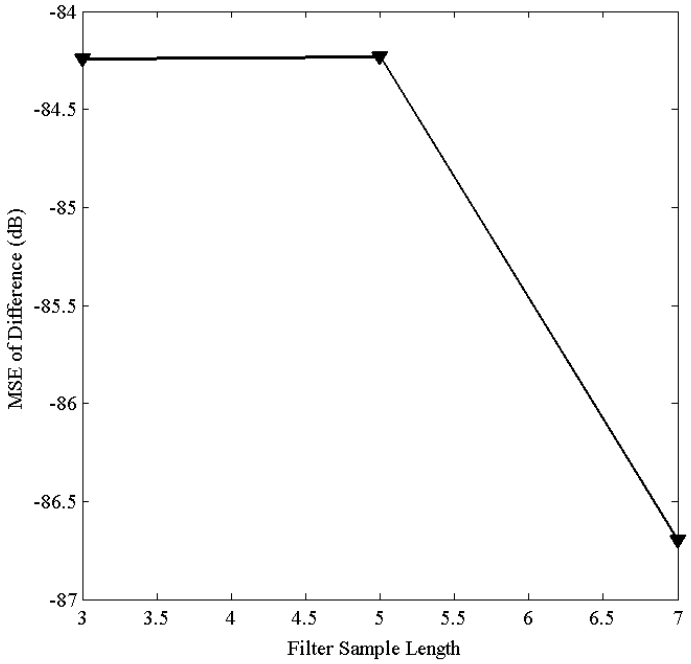


Figure 17. Mean squared error of the difference between filtered backprojection and 2-level recursion factorized backprojection vs. filter length.

The factorized backprojection function uses memory differently than the filtered backprojection. The recursive nature of the function increases the memory usage due to the fact that each level of recursion creates its own set of variables, and the parent levels are stored while the child levels are being processed.

At each level the amount of memory required is decreased due to the decimation of the phase history data and the image partitioning, but the recursion does increase the total amount of memory used. For zero levels of recursion, the memory is identical to the usage for filtered backprojection — 32MB. Increasing to one level of recursion, the memory usage with the simulation data set increases to 53MB. The general trend can be seen in Figure 18.

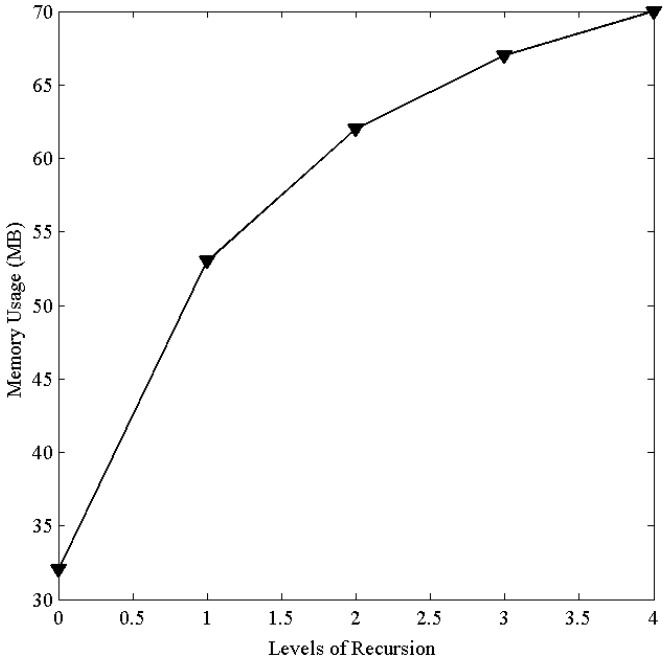


Figure 18. Memory usage versus levels of recursion for factorized backprojection.

Factorized backprojection has been shown to provide a speed increase, but it also comes with a penalty to image quality due to the aliasing issue. To counter the aliasing issue, an anti-aliasing filter can be applied for a very minor speed penalty.

The speed penalty is minor enough, and the improvement to image quality is great enough to warrant a longer filter. There is also a memory usage increase over what filtered backprojection uses. However, this memory increase has a limit and is not significant enough to detract from the speed increase afforded by using the factorized backprojection algorithm.

IV. Implementation on Raspberry Pi

Chapter 3 defined the broader design environment. The components that comprise the design environment include: a Raspberry Pi as the test platform, C++ as the programming language, and filtered and factorized backprojection as the algorithms implemented. Using a controlled experimental environment, Chapter 3 provided an ideal test base for the development of C++ code that is functional, but also works within the limited memory constraints supplied by a much smaller platform. It was possible to develop a fundamental code that maintained quality of the imagery while also decreasing run time. Now that all of these pieces are tested and accepted as a baseline, Chapter 4 will take the developed algorithm and actually apply it in more practical pretenses. This will serve as the first attempt to actually implement the designed code on to the smaller, low SWAP platform of the Raspberry Pi. The performance of the developed C++ code on the Raspberry Pi will be evaluated against the same criteria as used before: time to generate images, quality of images, and memory usage.

To evaluate the processing time, the filtered backprojection is first used to establish a baseline. The expectation for the processing time is that the filtered backprojection code will run slower on the Raspberry Pi due to its slower processor. On the laptop used to develop the code, the filtered backprojection code generated image data in 4 min 55 sec, while on the Raspberry Pi the image data was generated in 20 min 33 sec. These baseline results agree with the expectation that processor speed has an impact on the time to generate image data. Running the factorized backprojection code on the Raspberry Pi also shows an across the board slow down for all recursion levels. Figure 19 shows that the time to run trend observed on the development laptop is also observed on the Raspberry Pi, with the same minimum between 2 and 3 levels of recursion. These results show that, accounting for the slower processor of

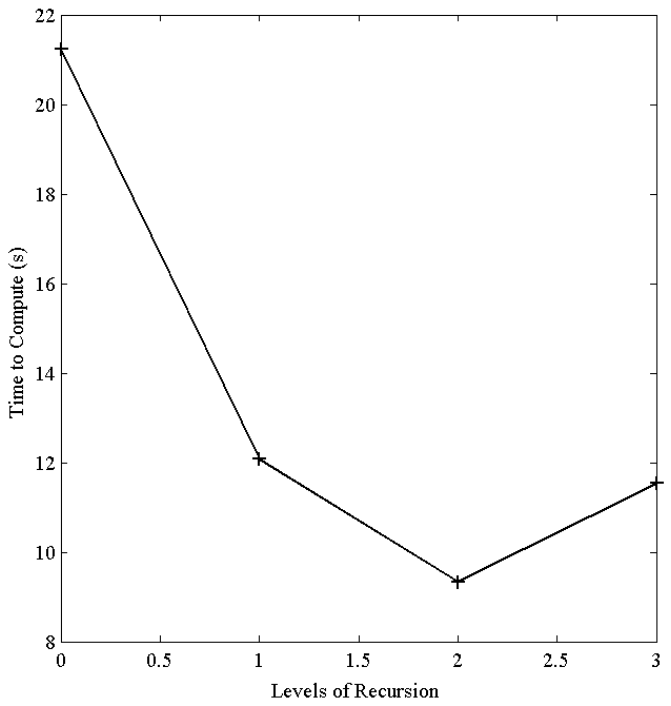
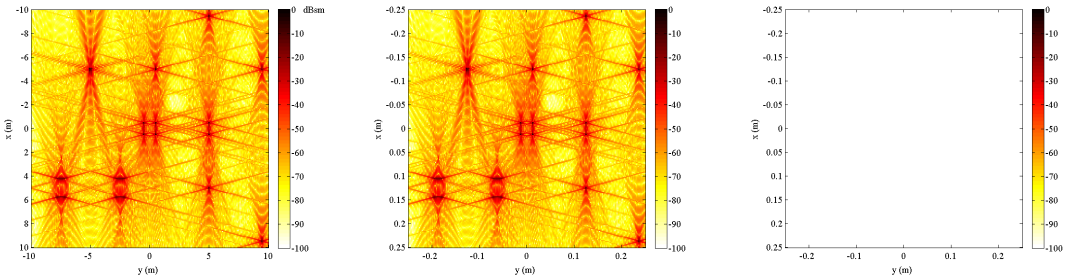


Figure 19. Time to compute vs. number of recursion levels on Raspberry Pi.

the Raspberry Pi, the speed performance of the algorithms remains comparable when implemented on the low cost, low SWAP platform.

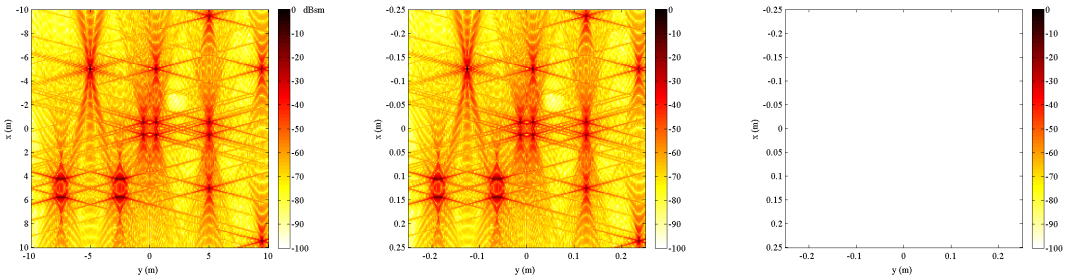
To compare the image quality, images generated on the Raspberry Pi are compared to equivalent images created on the development laptop. The expectation is that the platform for which the code is compiled for will have no effect on the quality of the images. Figures 20-23 show the images created on the development laptop on the left, images created on the Raspberry Pi in the middle, and the magnitude of the complex difference on the right. Visually the difference image appears to be blank. The actual maximum difference for each of the images is 0 for the precision of the data that was written to the image file. Since only the algorithm and system parameters (see Table 2) affect the image quality, the code is now only limited by the available compiler.

The results in Chapter 3 showed that the factorized backprojection algorithm uses an increasing amount of memory with each recursion level performed. On the



(a) C++ image on Laptop (b) C++ image on Raspberry Pi (c) Difference between image on laptop and Raspberry Pi.

Figure 20. Comparison of images generated on Development laptop vs. Raspberry Pi for zero recursion levels.

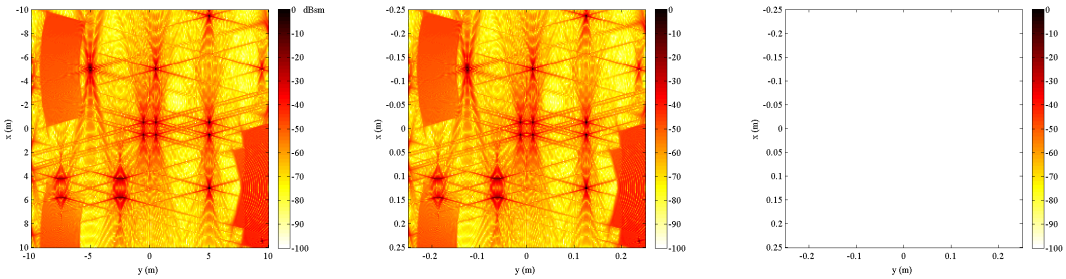


(a) C++ image on Laptop (b) C++ image on Raspberry Pi (c) Difference between image on laptop and Raspberry Pi.

Figure 21. Comparison of images generated on Development laptop vs. Raspberry Pi for one recursion level.

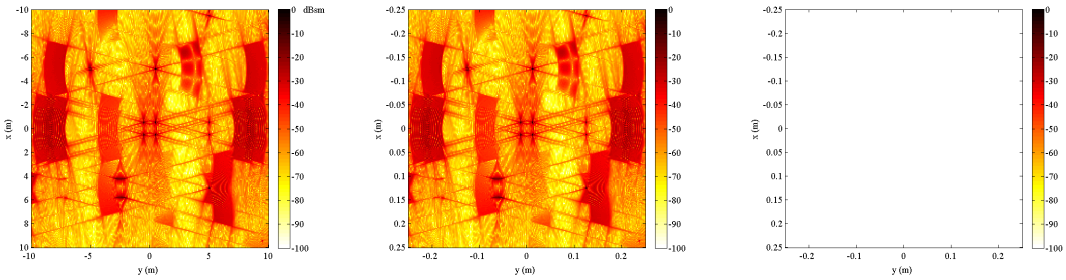
Raspberry Pi, memory is a very limited resource so any memory usage must be closely looked at. The memory growth seen on the Raspberry Pi matched the memory growth seen on the development laptop. For the parameters and image size from Table 2, the memory growth does not affect the time to generate image data on the Raspberry Pi. However, increasing any of the parameters that affect the amount of data to be processed (e.g. azimuth and frequency samples, or image size) will rapidly deplete the available memory on the Raspberry Pi.

Overall, the filtered and factorized backprojection algorithms performed as expected on the Raspberry Pi. Knowing that the Raspberry Pi has a slower processor



(a) C++ image on Laptop (b) C++ image on Raspberry Pi (c) Difference between image on laptop and Raspberry Pi.

Figure 22. Comparison of images generated on Development laptop vs. Raspberry Pi for two recursion levels.



(a) C++ image on Laptop (b) C++ image on Raspberry Pi (c) Difference between image on laptop and Raspberry Pi.

Figure 23. Comparison of images generated on Development laptop vs. Raspberry Pi for three recursion levels.

(700MHz versus 2.2GHz on the laptop) the speed was expected to be slower, though by how much was unknown.

V. Hardware Implementation Discussion

The factorized backprojection algorithm provides a significant decrease in processing time to create an image, as the results in Chapter 3 show. The processing time can be decreased further if some method of parallel computing is used. With a Raspberry Pi, the processor can only handle one process at a time (it is a single thread processor) and so cannot process in parallel. However, the Raspberry Pi has been proven to operate in what is called a Beowulf cluster [35]. A Beowulf cluster is a collection of identical compute systems, networked together and running a parallel programming software that allows each system in the cluster to share data and computation [35]. Connecting a number of Raspberry Pi systems in a Beowulf cluster would allow for a parallel approach to factorized backprojection using the Raspberry Pi platform.

The partitioning of the image in the factorized backprojection algorithm readily lends itself to parallel processing [36]. In a simple one recursion scenario, a "top" Raspberry Pi would filter and decimate the phase history data, divide the image up in to four quardants, and then send each quardant and a copy of the decimated phase history data to one of four connected Raspberry Pi's. The second tier Pi's would process their respective quadrant of the final image and then send that data back to the "top" Pi for reconstruction into the final image. Depending on the number of Raspberry Pi's available, the same process can be repeated for the second tier Pi's. Each of the second tier Pi's could filter and decimate the phase history data they received from the "top" tier, partition their quadrant and send the decimated phase history data and subimages to additional Pi's in a third tier for processing. The number of required processing units grows exponentially as more level of recursion are used, so the number of processing tiers has to remain small.

The Raspberry Pi also possesses a general processing unit (GPU) that can be

tapped into for processing. A GPU is typically used to process video data and has powerful parallel processing capability, but not usually accessible for normal operations. Currently, there are not many programming libraries available for using the GPU. One that is available is the GPU_FFT, a method of sending fast Fourier trasnform (FFT) operations from the FFTW library over to the GPU for processing [37]. If large FFT's (2^{20} or more elements) need to be calculated, it could be very adventageous to off-load that processing to the GPU.

VI. Conclusions/Future Work

The main goal of this thesis was to investigate the feasibility of implementing real time SAR imaging on a low cost, low SWAP platform. While a great deal of interest and research was spent on large-platform SAR, the use of small-platforms was limited to the civilian environment. However, smaller platforms do provide a useful tool that is lacking, and could open a new area of research. As very little data and research exists on this topic, this thesis set out to form the foundation and serve as a maiden test of feasibility for such a project. In order to pursue this goal, the project focused on a three-stage research process that advanced from the adaptation of SAR algorithm written using MATLAB, to one written in C++, a much less proprietary language that used less overhead memory to run. After adaptation of the code, stage one tested the script in C++ against the original MATLAB ran script. Testing for image quality and run time, both languages preformed comparatively. However, C++ did use less memory to execute the code that MATLAB did, as MATLAB is a much larger and diverse program.

The second stage of the project then took the adapted code and evaluated the ability to decrease run time without decreasing image quality. Using factorized back-projection, the first test of this stage did result in decreased run time, but poor image quality. As discussed by sampling theory in Chapter 2, a filter was added to the code to aid in maintaining image quality. The experiment then evaluated the length of filter in relation to the decrease in run time. While the filter resulted in very marginal increases in run time in comparison to the pre-filter model, the image quality was greatly improved. As such, it was concluded that the longer the filter, the greater the maintenance of image quality and the run time increase was minimal.

Lastly, the third stage transferred the experimental algorithm to the performance stage. The algorithm was then loaded and tested on a very basic small-platform

(Raspberry Pi), which served as a baseline, cost-effective, low-power platform. While the test did provide positive results as the code did run with satisfactory image quality, areas of memory usage and processor size greatly affected the run time of the Raspberry Pi.

Based on the conclusions gathered from the project, the overall goal of the thesis remained intact. It is feasible to run a SAR algorithm off of a low-cost, low-SWAP platform, though not practical or in real time with the current implementation. The research conclusions from each of the individual stages provided numerous areas for improvement and further research. Alternative implementations of the code could serve as an area of possible improvement for overall memory usage and memory efficiency, as well as the exploration and evaluation of other coding languages. As discussed back in Chapter 3, both C and Python could also serve as possible languages that would be worthy of testing and comparing to the C++ code that was adapted in this thesis. The code comparisons may also aid in decreasing run time, as well as the overall size of the processor in the given small platform. A small platform that may be a little pricier than the basic Raspberry Pi, or even a newer version of the Raspberry Pi could be better equipped to run the algorithm and therefore decrease the total run time of the code.

Another avenue for research and development would be to expand on the code. This could include features that reads data directly in from a given device, as opposed to reading in from a specific file as the code currently does. This read-in would serve as a very interesting alternative worthy of evaluation . At the output end, the data could directly output to a video device as it is created: currently the image data is saved to a file for later conversion to an image. If the future goal of this project would be truly real time image creation, both of these options would be necessary to receive radar data and create an image for immediate use.

There are many new research questions and project possibilities that this thesis has opened up for future AFIT students and analysts. While the project is still very much in its infancy, there were promising steps that show promise in researching areas of small-platform SAR for future tactical imagery.

Bibliography

1. C. Sherwin, J. Ruina, and R. Rawcliffe, "Some Early Developments in Synthetic Aperture Radar Systems," *IRE Transactions on Military Electronics*, vol. MIL-6, no. 2, pp. 111–115, April 1962.

2. W. G. Carrara, R. S. Goodman, and R. M. Majewski, *Spotlight Synthetic Aperture Radar Signal Processing Algorithms*. Norwood, MA: Artech House, 1995.

3. C. V. Jakowatz, D. E. Wahl, P. H. Eichel, D. C. Ghiglia, and P. A. Thompson, *Spotlight-Mode Synthetic Aperture Radar: A Signal Processing Approach*. New York, NY: Springer Science and Business Media, LLC, 1996.

4. V. Vu, T. Sjogren, and M. Pettersson, "Fast factorized backprojection algorithm for UWB SAR image reconstruction," in *IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, July 2011, pp. 4237–4240.

5. J. Munson, D.C., J. O'Brien, and W. Jenkins, "A Tomographic Formulation of Spotlight-Mode Synthetic Aperture Radar," *Proceedings of the IEEE*, vol. 71, no. 8, pp. 917–925, Aug 1983.

6. Z. Li, J. Wang, and Q. H. Liu, "Frequency-Domain Backprojection Algorithm for Synthetic Aperture Radar Imaging," *IEEE Geoscience and Remote Sensing Letters*, vol. 12, no. 4, pp. 905–909, April 2015.

7. L. Ulander, H. Hellsten, and G. Stenstrom, "Synthetic-Aperture Radar Processing Using Fast Factorized Back-Projection," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 39, no. 3, pp. 760–776, July 2003.

8. A. Yegulalp, "Fast Backprojection Algorithm for Synthetic Aperture Radar," in *The Record of the 1999 IEEE Radar Conference*, 1999, pp. 60–65.

9. S. Xiao, J. Munson, D.C., S. Basu, and Y. Bresler, "An $N^2 logN$ Back-Projection Algorithm for SAR Image Formation," in *Conference Record of the Thirty-Fourth Asilomar Conference on Signals, Systems and Computers*, vol. 1, Oct 2000, pp. 3–7 vol.1.

10. S. Basu and Y. Bresler, "$\mathcal{O}(N^2 \log_2 N)$ filtered backprojection reconstruction algorithm for tomography," *IEEE Transactions on Image Processing*, vol. 9, no. 10, pp. 1760–1773, Oct 2000.

11. A. Boag, Y. Bresler, and E. Michielssen, "A multilevel domain decomposition algorithm for fast $\mathcal{O}(N^2 \log N)$ reprojection of tomographic images," *IEEE Transactions on Image Processing*, vol. 9, no. 9, pp. 1573–1582, Sep 2000.

12. Y. Ding and J. Munson, D.C., "A Fast Back-Projection Algorithm for Bistatic SAR Imaging," in *Proceedings of the International Conference on Image Processing*, vol. 2, 2002, pp. II–449–II–452 vol.2.

13. T. Pipatsrisawat, A. Gacic, F. Franchetti, M. Puschel, and J. Moura, "Performance Analysis of the Filtered Backprojection Image Reconstruction Algorithms," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, March 2005, pp. v/153–v/156 Vol. 5.

14. A. Ribalta, "Optimizing the Factorisation Parameters in the Fast Factorized Backprojection Algorithm," in *9th European Conference on Synthetic Aperture Radar (EUSAR)*, April 2012, pp. 356–359.

15. L. M. H. Ulander, P.-O. Froelind, A. Gustavsson, D. Murdin, and G. Stenstroem, "Fast Factorized Back-Projection for Bistatic SAR Processing," in *8th European Conference on Synthetic Aperture Radar (EUSAR)*, June 2010, pp. 1–4.

16. G. Shippey, S. Banks, and J. Pihl, "SAS image reconstruction using Fast Polar Back Projection: comparisons with Fast Factored Back Projection and Fourier-domain imaging," in *Oceans 2005 - Europe*, vol. 1, June 2005, pp. 96–101 Vol. 1.

17. P. Frolind and L. Ulander, "Evaluation of angular interpolation kernels in fast back-projection SAR processing," *IEE Proceedings - Radar, Sonar and Navigation*, vol. 153, no. 3, pp. 243–249, June 2006.

18. S. Banks and H. Griffiths, "The use of Fast Factorised Bank-Protection for synthetic aperture sonar imaging," 2002.

19. E. Zaugg, M. Edwards, and A. Margulis, "The SlimSAR: A Small, Multi-Frequency, Synthetic Aperture Radar for UAS Operation," in *2010 IEEE Radar Conference*, May 2010, pp. 277–282.

20. J. A. Jackson, "EENG714 Advanced Radar Systems Class Notes," Course Handout, Air Force Institute of Technology, Wright-Patterson AFB OH, Summer Quarter 2015.

21. N. J. Willis and H. D. Griffiths, *Advances in Bistatic Radar*. Edison, NJ: SciTech Publishing, 2007.

22. A. J. Jerri, "The Shannon sampling theorem — Its various extensions and applications: A tutorial review," *Proceedings of the IEEE*, vol. 65, no. 11, pp. 1565–1596, 1977.

23. I. G. Cumming and F. H. Wong, *Digital Signal Processing of Synthetic Aperture Radar Data*. Norwood, MA: Artech House, Inc., 2005.

24. A. Brandt, J. Mann, M. Brodski, and M. Galun, "A Fast and Accurate Multilevel Inversion of the Radon Transform," *SIAM Journal on Applied Mathematics*, vol. 60, no. 2, pp. 437–462, 2000. [Online]. Available: http://dx.doi.org/10.1137/S003613999732425X

25. A. George and Y. Bresler, "Fast Tomographic Reconstruction via Rotation-Based Hierarchical Backprojection," *SIAM Journal on Applied Mathematics*, vol. 68, no. 2, pp. 574–597, 2007. [Online]. Available: http://dx.doi.org/10.1137/060668614

26. Majumder, U. K. and Casteel Jr, C. H. and Buxa, P. and Minardi, M. J. and Zelnio, E. G. and Nehrbass, J. W., "SAR data exploitation: computational technology enabling SAR ATR algorithm development," in *Defense and Security Symposium*. International Society for Optics and Photonics, 2007, pp. 65 680L–65 680L.

27. Hellsten, H. and Ulander, L. M. and Gustavsson, A. and Larsson, B., "Development of VHF carabas II SAR," in *Aerospace/Defense Sensing and Controls*. International Society for Optics and Photonics, 1996, pp. 48–60.

28. Raspberry Pi Foundation. (2016) Raspberry Pi Model Specifications. [Online]. Available: https://www.raspberrypi.org/documentation/hardware/raspberrypi/models/specs.md

29. Raspbian. (2016) FrontPage - Raspbian. [Online]. Available: https://www.raspbian.org/

30. MathWorks Inc. (2016) MathWorks – MATLAB and Simulink for Technical Computing. [Online]. Available: http://www.mathworks.com/

31. Python.org. (2016) Welcome to Python.org. [Online]. Available: http://www.python.org/

32. L. A. Gorham and L. J. Moore, "SAR image formation toolbox for MATLAB," in *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2010, pp. 769 906–769 906.

33. J. A. Jackson, "MATLAB Bistatic Backprojection Script," Course Handout, Air Force Institute of Technology, Wright-Patterson AFB OH, Summer Quarter 2015.

34. A. Oppenheim and R. Schafer, *Digital Signal Processing*. Prentice-Hall, 1975.

35. J. Kiepert, "Creating a raspberry pi-based beowulf cluster," 2013.

36. A. Rogan and R. Carande, "Improving the fast back projection algorithm through massive parallelizations," pp. 76 690I–76 690I–8, 2010. [Online]. Available: http://dx.doi.org/10.1117/12.850332

37. Raspberry Pi Foundation. (2016) Raspberry Pi Foundation. [Online]. Available: https://www.raspberrypi.org/

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704–0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
24–03–2016	Master's Thesis	Sep 2014 — Mar 2016

4. TITLE AND SUBTITLE

Implementation and Performance
of Factorized Backprojection
on Low-cost Commercial-Off-The-Shelf Hardware

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Rasmussen, Alec S., Capt, USAF

5d. PROJECT NUMBER

16G217

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-16-M-041

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
Sensors Directorate
WPAFB OH 45433
COMM 937-528-8940
Email: kung.ding@us.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

RYMH

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Traditional Synthetic Aperture Radar (SAR) systems are large, complex, and expensive platforms that require significant resources to operate. The size and cost of the platforms limits the potential uses of SAR to strategic level intelligence gathering or large budget research efforts. The purpose of this thesis is to implement the factorized backprojection SAR image processing algorithm in the C++ programming language and test the code's performance on a low cost, low size, weight, and power (SWAP) computer : a Raspberry Pi Model B. For a comparison of performance, a baseline implementation of filtered backprojection is adapted to C++ from pre-existing MATLAB® code. The factorized backprojection algorithm shows a computational improvement factor of 2-3 compared to filtered backprojection. Execution on a single Raspberry Pi is too slow for real-time imaging. However, factorized backprojection is easily parallelized, and we include a discussion of parallel implementation across multiple Pis.

15. SUBJECT TERMS

Backprojection, SAR, Raspberry Pi, Factorized backprojection

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Julie Jackson, AFIT/ENG
U	U	U	U	80	19b. TELEPHONE NUMBER *(include area code)* 937-255-3636 x4678; Julie.Jackson@afit.edu