Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Engineering

Electronic Engineering

MERIT.jl: Julia's Version

Aaron Dinesh

Supervisor: Associate Prof. Declan O'Loughlin

April 9, 2024

A Final Year Project submitted in partial fulfilment
of the requirements for the degree of
MAI (Electronic and Computer Engineering)

Declaration

I hereby declare that this Final Year Project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

I consent/do not consent to the examiner retaining a copy of the thesis beyond the examining period, should they so wish (EU GDPR May 2018).

I agree that this thesis will not be publicly available, but will be available to TCD staff and students in the University's open access institutional repository on the Trinity domain only, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgment. **Please consult with your supervisor on this last item before agreeing, and delete if you do not consent**

Signed: Aaron Dinesh Date: April 9, 2024

Abstract

MERIT aims to provide a software framework that is robust, easy to use and performant. It implements a variety of microwave imaging algorithms and a myriad of helper functions, all while leveraging the powerful features available in Julia. MERIT.jl also implements a "Scan" abstract datatype which allows users to subtype their own specialized datatype. Organizing the datatypes in this way means that MERIT.jl plays very well with Julia's own type hierarchy and also the other language features that depend on this. To encourage type safety, MERIT.jl implements a lightweight Points class which allows for efficient processing of coordinate points. In this way, collections of points won't simply be a matrix of Floats or Ints instead, they would be a Vector of the Points type. In this way, the Julia compiler will throw an error when Points aren't passed in the right argument, instead of providing a wrong output.

Acknowledgements

Thanks, Everyone!

List of Figures

Contents

Introduction

The microwave imaging field has seen a rising interest in the medical field evidenced by the numerous clinical trials which are being conducted by research groups around the world [1,2,4]. A cursory search on GitHub for software around microwave imaging yielded few useful results, with many either being specialized repositories for a particular task or performing some machine learning analysis on microwave data. Only one repository stood out as a generalized library that gives researchers all the tools needed to easily test different algorithms; the MERIT toolbox developed by Prof O'Loughlin, M. A. Elahi, E. Porter, et. al [**?**]. Other fields of science have seen numerous benefits from the introduction of comprehensive open-source libraries. Libraries such as PsychoPy and PsyToolkit have allowed researchers to design and conduct experiments in a matter of hours by packaging common functions in an easy-to-use library [5]. With the rise in the number of systems that can perform microwave imaging and the vast amounts of data being generated from these systems, it is imperative that there are a variety of toolkits available to not only analyze data from these current systems but also from future systems. This BAI project aims to consider the following questions:

- Improving the accessibility of Microwave Imaging
- Increasing the compatibility between systems, the data these systems generate and the software used to analyze this data
- Creating an intuitive, easily extensible and customizable library
- Leveraging the features of a coding language to create a performant library

MERIT.jl was motivated by a need to increase the interoperability between the data produced from systems and the programs that would be able to process them. Currently, any researcher who wants to design a novel microwave imaging algorithm or test the efficacy of their new systems has to spend time and effort creating entirely new software suites. More often than not these programs either tend to be closed source, or not compatible with other systems mainly because there are not many incentives for a researcher to invest the time required in creating a truly configurable and extensible open-source system. With the rise and success of clinical trials in microwave imaging, this problem is only going to get worse as more and more systems are developed.

The use of such open-source software has seen widespread use in a myriad of fields. PsychoPy and PsyToolkit are two such frameworks that revolutionized the field of psychological sciences. By implementing commonly used functions and scripts, they have allowed researchers to design and run experiments in a matter of hours. It has also allowed researchers who have little to no programming experience to get up and running with automated data processing, thereby

allowing them to focus more on the quality of their experiment [5].

That is where this project comes in. MERIT.jl aims to be an easy-to-use, extensible and featureful library. The goal being that anyone, regardless of coding experience, would be able to quickly create a script that allows them to process and visualize the scan data they have collected, as well as allowing more experienced coders to develop and test new algorithms with ease. The following sections will contain:

- A literature review on the existing microwave imaging systems
- A look into existing microwave imaging frameworks
- A discussion about the design choices and Julia features that are included
- An examination of the results and possible future work

The source code and the library are hosted on GitHub for anyone to view and amend: `https://github.com/AaronDinesh/MERIT.jl`

Background

1.1 Literature Review

In order to understand the design requirements of the library, a literature review needed to be performed. Due to the relative infancy of the field, there are various competing configurations with no clear winner in sight. However, some patterns can still be gleaned from current clinical trials which are broadly representative of the direction the field is moving in. In all of these systems, the patient to be scanned would lie in the prone position on an imaging table. The patient would then pass their breast through a hole in the table into some type of imaging apparatus. The imaging apparatus varies slightly from system to system, but they all contain some type of antenna array that is used for the imaging process. Most systems will adopt one of the following antenna setups

1. Monostatic
2. Leveled Multistatic
3. Fully Multistatic

These terms relate to the position of antennas around the breast tissue and how the resulting scan data would be structured.

1.1.1 MARIA M4

The first system we will look at is the MARIA M4 system developed by Preece et al within the Electrical and Engineering Department of the University of Bristol [1]. This is the 4th iteration in a series of MARIA systems that evolved from a system of 16 UWB antennas to 60 antennas in the current system. These all operate in a multistatic configuration, meaning that any antenna in the array can listen to any other antenna in the array, an example of which can be seen in Figure 1.1. This figure shows a top-down view, however, one can imagine this being generalized to a hemisphere of antennas around the breast.

As stated before, the MARIA M4 system makes use of the UWB spectrum over a frequency range of 3.0 to 8.0 GHz. A commercial-grade Vector Network Analyzer (VNA) was used as the system signal source. The VNA, operating in a stepped continuous wave mode, was used to both produce the transmitted sine waves and record the frequency and phase of the reflected waves at the receiving antennas. The choice of a commercial-grade VNA is indicative of the prototype nature of the MARIA M4 system, where easy reconfigurability of the frequency imaging range would be of higher importance than price. While not explicitly stated, one would presume that the M5 system would replace the VNA with some application specific

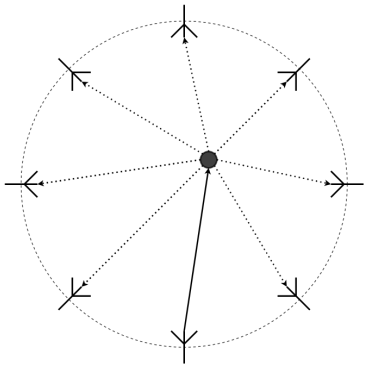


Figure 1.1: Example of a Fully Multistatic Configuration (Top-Down)

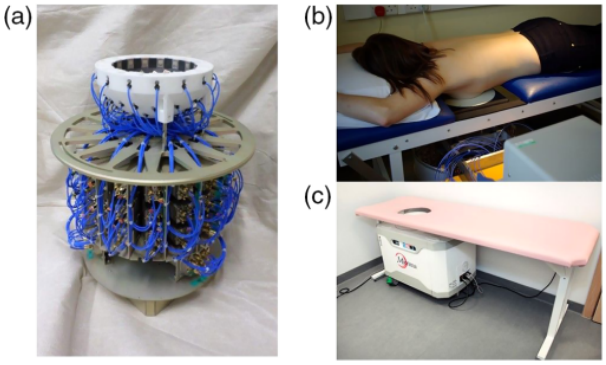


Figure 1.2: The MARIA M4 and M5 system. (a) The MARIA M4 antenna array. (b) The M4 in a clinical setting. (c) The integrated M5 package [1]

hardware to reduce the overall cost of the system. The M4 system exploits the inherent symmetry in the antenna reciprocity to halve the number of channels (made of a transmitting and receiving antenna) collected, thereby speeding up the scan time. For the MARIA M4 system, this equates to a 1770 reduction in the number of channels collected. Figure 1.2, shows the antenna array used in the M4 system (a), as well as the M5 system (c) which is an integrated package.

The team also conducted a clinical study in order to test the efficacy in women who already attend symptomatic breast care clinics. In total 86 patients were included with ages ranging from 24 - 78 years old; the mean age being 51.4. The inclusion criteria for the study required that possible participants be clinically symptomatic, be able to be imaged by ultrasound and mammograms (these being the control), be able to lie in the prone position and have cup sizes within 310 to 850 ml. The types of lesions included in the study were mostly cysts and cancers, but some other conditions such as hematoma, lipoma and fibroadenoma were also included. The goal of the study was to test the sensitivity of the M4 system; the sensitivity metric being determined based on the ability of the system to localize a lesion as it correlated with the location in the ultrasound image. The M4 system showed a sensitivity of 74% (64/86) when compared with the "gold-standard' of an ultrasound. The research team also divided the group into pre-/peri- and post- menopausal women and found that the sensitivity was 75% and 73% respectively. However, the reliability of these results are called into question when considering the sample size of the study. Given a sample size of 86, assuming a normal distribution and that the results are statistically significant ($p < 0.05$, $Z = 1.96$), a 7.11% margin of error was calculated. While this may not be enough to conclusively prove that the

Figure 1.3: The Wavelia System [2]

M4 system is a viable alternative to Mammograms, it is enough to show promise. With a larger sample size, this margin of error could be narrowed further.

1.1.2 Wavelia

The second system that was considered was the Wavelia Microwave Breast Imaging System developed by MVG Industries [2]. The Wavelia system integrates the imaging system as well as the examination bed into one complete package (Figure 1.3). The integrated package makes the Wavelia system an appealing choice for some hospitals, however, its large size may be a barrier to adoption in some facilities where space is a premium.

Like in the MARIA M4 system, patients lie in a prone position and place their breasts in the circular cutouts on the bed. The system then begins to create a 3D reconstruction of the exterior of the breast using a stereoscopic camera. This also allows for the breast volume to be explicitly calculated rather than being inferred like in the MARIA system. The Wavelia makes use of the UWB spectrum when imaging the breast, although they opt for a narrower frequency range of 0.5 - 4.0 GHz compared to the 3.0 - 8.0 GHz range of the MARIA system. The antenna configuration, unlike the MARIA system, is an array of 18 Vivaldi-type probes arranged in a concyclic manner on a horizontal plane. These antennas operate in a Multistatic manner and image the breast in sections parallel to the coronal plane. The entire antenna assembly moves downwards in 5mm intervals to image the entire breast (Figure 1.4). This is a leveled multistatic system as opposed to the fully multistatic system in the MARIA M4. This leveled multistatic approach has the benefit of a theoretically infinite vertical resolution. If the radiologist wanted a finer resolution along the coronal plane, they would just have to tweak the array vertical step size, rather than having to manufacture an entirely new antenna array, like in the MERIT system. This leveled approach also allows the parameters of the reconstruction algorithm to be easily changed based on the particular section of the breast that we are imaging. The fully multistatic approach, however, would need significant additional logic in the post-processing steps to determine which channels are coplanar with a particular section.

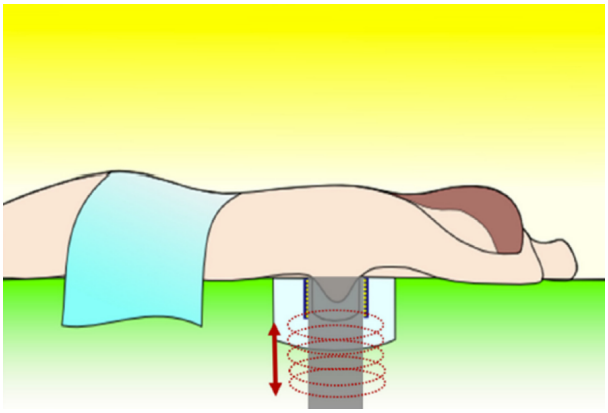


Figure 1.4: The Leveled Multistatic Approach of Wavelia [2]

The Wavelia paper [2] conducted a feasibility study on 25 female participants who were recruited after presenting with symptoms to the Symptomatic Breast Unit at Galway University Hospital, Ireland. Their inclusion criteria required that participants to have a mammogram that was performed at the time of symptomatic presentation to the hospital, be capable of lying in the prone position for a length of 15 minutes, have a bra size larger than 32B and a breast size equivalent to a B cup. The final criteria required that the participant's submerged breast have enough margin between the breast and the cylindrical container in order to accommodate the transition liquid. This was determined by the clinician at the time of the trial. Out of the 25 patients who presented with a palpable lump, one patient's scan had to be removed, due to the lump being later classified as normal tissue. 11 of the participants had a biopsy confirmed carcinoma and out of these, the Walia system detected 9 lesions, with 7 being located to the appropriate region. Overall the system detected an abnormality in 21 of the 24 participants, leading to a sensitivity of 87.5%. The researchers do note some limitations of the Walia system, namely it can't detect any lesions smaller than 10mm. This is significant since the size of the detected lesion plays a big factor when deciding whether a lesion is cancerous or not. Another limitation of the system is that breast sizes that are too small cannot be scanned in any great detail by the Wavelia system. Due to the patients being in the prone position, their breast tissue needs to hang down far enough to have multiple sections of their breast be imaged by the antenna array. The researchers are working on a subsequent system that should address all the aforementioned limitations. Overall the participants had a positive outlook on the system. 23 out of the 25 women said that they would recommend the procedure to other women and all of the women agreed that the information provided was clear and well understood.

1.1.3 TSAR

The third and final system considered for the project was the TSAR system [4]. Standing for Tissue Sensing Adaptive Radar, this system was developed by the University of Calgary to address some of the shortcomings of the previous two systems. In the MERIT and Wavelia systems, reflections from the skin can muddy the data and can lead to artifacts in the final image. In order to get around this, both systems take an additional scan, offset by a fixed rotational amount in the coronal plane. The idea being, that any reflections that appear on the first scan would appear with similar amplitude in the second scan at the same time position,

Figure 1.5: The TSAR Prototype [3]

while the tumor would show in a different place, provided that the tumor doesn't lie on the axis of rotation. This method is known as Rotational Subtraction [6]. The TSAR system, on the other hand, makes use of an adaptive algorithm that instead estimates the skin response at an antenna as the weighted sum of the responses from the neighboring antennas. This skin response can then be subtracted from the current antenna to remove the reflection artifacts [7].

Like the previous two systems, patients lie in the prone position on the examination table, with their breast being submerged in an immersion liquid. However, unlike the previous two systems, the TSAR operates in a monostatic configuration. In this setup, there is only one antenna that acts as both the transmitting and the receiving antenna. This antenna, usually fixed on some type of rotating apparatus, would rotate around the breast to image a section. It would then step vertically by a fixed amount and repeat the previous step, eventually imaging the entire breast. The TSAR system also operates on a much wider section of the UWB spectrum, from 50 MHz to 15 GHz, with the frequency data being collected by a VNA. The system also makes use of a laser in order to help with the 3D reconstruction of the breast during post-processing. The prototype setup can be seen in Figure 1.5

The benefit of the monostatic configuration is that the number of antenna locations per row and the number of rows are parameters that can be tweaked depending on whether the radiologist would want a quicker scan or higher precision. One can imagine the scenario of a rapid screening center as part of a national screening program where the speed of the scan is valued over the precision of the scan. As such this provides a notable benefit over the previous two systems. The clinical trial conducted by this study was extremely limited, only including 8 successfully imaged patients. Due to the low sample size, one can't reliably say whether this system and its adaptive reflection-suppressing algorithm can outperform the other two systems in terms of accuracy, although the TSAR system does show promise.

1.1.4 Beamformers

Without loss of generality, imagine a line array of wave emitters. If all of these begin to emit at the same time, the individual waves would constructively and destructively interfere with each other as their peaks and troughs align and misalign. This property can be exploited in such a way so that the waves constructively interfere in one direction and destructively interfere in all the other directions, effectively aiming the beam in a particular direction. A diagram of this can be seen in Figure 1.6.

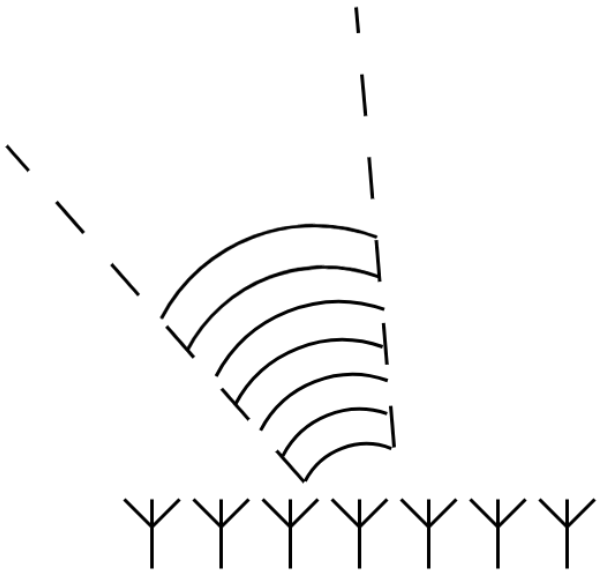


Figure 1.6: Phased Array Diagram

This is the forward beamforming process, but one can also consider the reverse process. Considering Figure 1.7 imagine a point emitter embedded in a 2D plane, that radiates circular waves evenly in all directions. Now imagine a line array that receives this wave. Due to the diffusion of the wave through space, the same wave will appear at each antenna at different times. This would manifest as the same amplitude appearing at different times, in the signal graphs, even though the impulses come from the same wave. Reverse beamforming then, is the process of varying the phase and amplitude of the received signals in order to estimate the intensity at the location where the wave originated. Inverse beamforming is a process used in many fields such as radio astronomy and seismic imaging, so there already is a wealth of research and plenty of algorithms to choose from. The rest of this section will talk about the various beamformers that are popular in the field of microwave imaging.

Delay and Sum

The Delay and Sum (DAS) beamformer is the template for most other beamforming algorithms in the field. The DAS beamformer posits that every antenna has recorded the same source and that the delay in each signal is due to the relative distance between the receiving antenna ($\mathcal{A}'$) and the transmitting antenna ($\mathcal{A}$). As such, if one was to delay the signals by the correct amount, and sum over all the received signals, one would be able to estimate the energy at the source. This same idea can be applied to signals received from the aforementioned imaging systems. Since these work in the frequency domain, the following explanations and equations

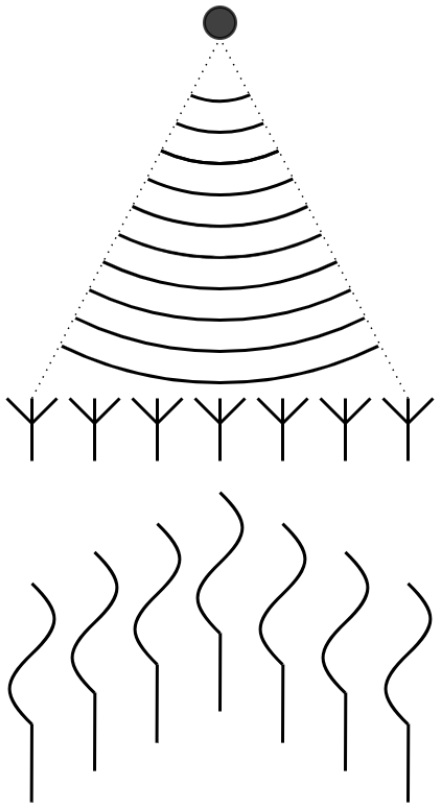


Figure 1.7: Point Emitter with a Phased Array

will work on this assumption, but these equations can easily be converted to the relative time domain functions. As stated before a point from the imaging domain is chosen (r). We then estimate the path delay of the wave from the transmitting antenna to the receiving antenna along the path.

$$\tau_{\mathcal{A}'_i,\mathcal{A}_j}(r) = \frac{\sqrt{\epsilon}}{c_0}\left[\|\mathcal{A}' - r\| + \|r - \mathcal{A}\|\right] \tag{1.1}$$

ϵ is the relative permittivity of the medium. In reality, this value changes from patient to patient and even varies within the breast of each patient depending on the path taken. However in order to achieve a practical beamformer, one must estimate an averaged relative permittivity value for the entire breast, we will label this as ϵ_i and it will parametrize the DAS beamformer. As well as this, equation 1.1 assumes a straight-line path between the point and the transmitting and receiving antennas. This is rarely the case, however, in practice making this assumption only yields a maximal error of 3mm in position while greatly simplifying the delay calculations [?]. Using this, the received signals are delayed and then summed across all stepped input frequencies. This result is then squared to give us the energy at the chosen point. Thus the point is imaged, This process is repeated for all the points in the imaging domain, overall the DAS beamforming equation can be represented as such:

$$I_{\epsilon_i}(r) = \left[\sum_{\Omega} \sum_{\mathcal{A}'} \sum_{\mathcal{A}} S_{a,a'}[\omega] e^{j\omega\tau_{\epsilon_i,a,a'}(r)} \right]^2 \tag{1.2}$$

The DAS algorithm was the chosen beamformer for the MERIT.jl library due to its simplicity to implement and the time constraints I had to work with. However, two other beamformers will be discussed below and will be compared with the DAS algorithm.

Weighted Delay and Sum Beamformer

The Weighted Delay and Sum (WDAS) Beamformer can be considered as a further generalization of the DAS beamformer. In equation 1.2 all channels contribute equally to the final result due to the implicit weighting factor of 1. The equation also assumes that all signals travel along similar paths, and therefore a constant speed (due to the fixed ϵ_i). In reality, this is only true for antennas that are closer together. The greater the distance between the transmitting and receiving antennas, the greater the chance, that the waves deviate from the straight line path, ergo the estimation of the speed and subsequently the delay along that path will be wrong. Essentially, the algorithm needs to have some mechanism that would penalize the signal from antennas that are further away and reward signals from antennas that are close by. This is what S.A. Shah Karam et al proposed in their 2021 paper, "Weighted delay-and-sum beamformer for breast cancer detection using microwave imaging" [?]. They define a weighting factor based on the transmitter-receiver distance (TRD_i) for the t^{th} observation:

$$w_i = \alpha - |r_{T_{r_i}} - r_{R_{c_i}}| \tag{1.3}$$

In order to reward signals from antennas that are close and penalize antennas that are far away one must subtract the TRD_i from a positive constant, α. This parameter is patient-specific and must be changed based on the homo- or heterogeneity of the breast tissue. The paper used an α of 20cm for their results. One important thing to note is that these weighting factors are data-independent. Since the TRD_i relies only on the distance between the antennas, these weighting factors can be computed beforehand. Another thing of note is that these weighting factors are independent of the chosen focal point as well, which means we can compute a normalized weight and apply it to the signal before employing the traditional DAS algorithm. The normalized weighting factor $\hat{w}_i$ is calculated as follows, where M is the number of channels:

$$\hat{w}_i = \frac{w_i}{\sum_{i=0}^{M} w_i} \tag{1.4}$$

Delay Multiply and Sum

The Delay Multiply and Sum (DMAS) algorithm was first proposed by Hooi Been Lim et al in 2008 [?]. Instead of considering each channel in isolation across the frequency bands, the DMAS algorithm proposes a pair-wise multiplication of the delayed signals before summing across the frequency range. For a focal point r the equation would be as follows:

$$I_{\epsilon_i}(r) = \sum_{\Omega} \sum_{\mathcal{A}'} \sum_{\mathcal{A}} \sum_{\mathcal{B}'} \sum_{\mathcal{B}} S_{a,a'}[\omega] e^{j\omega\tau_{\epsilon_i,a,a'}(r)} S_{b,b'}[\omega] e^{j\omega\tau_{\epsilon_i,b,b'}(r)} \tag{1.5}$$

Here $\mathcal{A}' = \mathcal{B}''$ are the receiving antennas and $\mathcal{A} = \mathcal{B}$ are the transmitting antennas. The paper then compared the DMAS algorithm with the traditional DAS algorithm and found that the pairwise multiplication of the delayed signals before summation provided greater contrast and sharper results, greatly increasing the SCR of the image. Figure 1.8 shows the differences between the DAS and DMAS algorithms and most importantly, how the DMAS beamformer provides a greatly reduced noise floor.

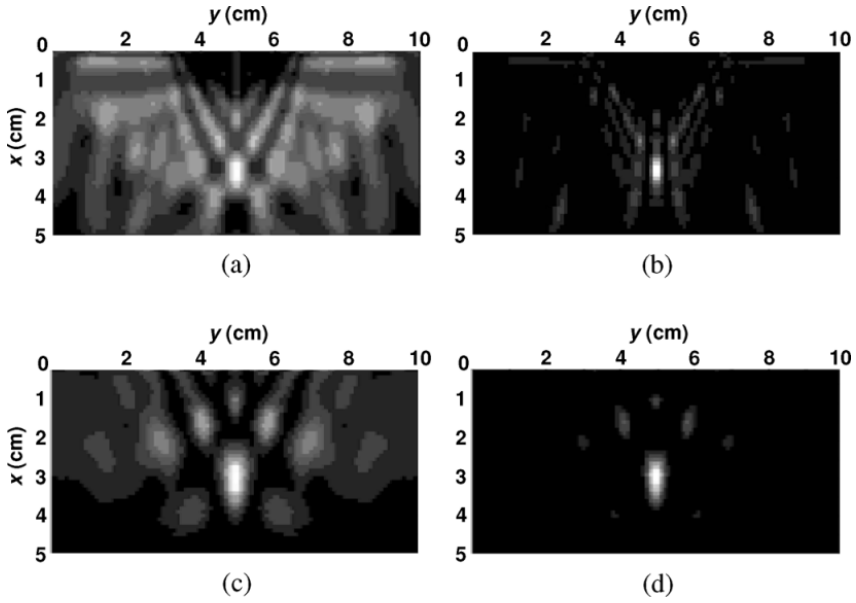


Figure 1.8: Comparason between DAS (a, c) and DMAS (b, d) Beamformers

The original paper never provided an explanation for the surprising effectiveness of the algorithm, leaving the explanation for a future paper that they never wrote. However, this did not stop the wider research community from providing possible explanations as to how the DMAS algorithm provides the results it does. One explanation provided by Prof O'Loughlin posits that the pairwise multiplication rewards signals that have a high degree of coherency while disproportionately penalizing incoherent signals. In essence, the coherent signals get brightened a lot more than the incoherent signals would [?]. G. Matrone et al backs up this idea by noting that the DMAS beamformer, after the signals are time aligned like in DAS, is the autocorrelation function of the receiver antenna with the auto terms excluded [?].

Julia

2.1 Why Julia?

One of the goals of MERIT.jl was to streamline the development process for new imaging algorithms. As well as this MERIT.jl aims to lower the barrier of entry and the coding knowledge needed in order to set up these data processing and visualizing pipelines. In order to fulfill these dreams, the right coding language needed to be selected. Since the barrier of entry needs to be low, a high-level language needs to be selected and since the primary target of MERIT.jl are researchers, it needs to be a language that they are familiar with, or one which they can easily learn. If the language chosen is too difficult to learn, any potential time savings from the ease of use would be heavily outweighed by the time spent learning the language.

2.1.1 Python

Python was one of the languages considered. Python's philosophy was code readability over all and this fact is complimented by its high-level syntax and indentation requirements. It was first introduced towards the end of the 1980's by Guido van Rossum and has had an expansive community ever since. The Python package repository and pip have allowed the community to develop libraries that can be easily shared and downloaded. All these factors catapulted Python into the limelight and it has become the go-to language for any data science and deep learning application. However, several drawbacks severely limit the usability of Python for high performance applications. Firstly, Python is an interpreted language, meaning that at runtime, the Python interpreter reads the Python file line by line and calls the relevant machine code. Due to this interpretation step, raw Python code is slow to run. Due to this reason, most performant libraries in Python tend to use C-optimized implementations behind the scenes. So in order to develop any decently performing Python libraries, one also has to be proficient in C as well as know how to create C and Python bindings. Secondly, the Python Interpreter makes use of a runtime lock known as the Global Interpreter Lock (GIL) which makes parallel processing in Python incredibly difficult. The reason for the GIL comes down to Python's use of reference counting for memory management. Reference counting is a method where each object in Python gets assigned a reference variable that keeps track of the number of references that point to that object. As references to the object get created, the count goes up, as references get deleted or reach the end of their scope, the count goes down. When the reference variable reaches zero, this becomes a flag for the object to be deleted, since there are no longer any references pointing to this object. In some threaded applications, this can cause some issues where the reference count gets updated by multiple threads simultaneously. In some cases, this can cause the reference count to never reach zero

leading to a memory leak, or it can reach zero too early and the object gets removed. To get around this, Python puts a lock on the interpreter itself, this is the GIL. This means that any Python bytecode needs to first acquire a lock on the interpreter before it can be executed. This makes multithreading in Python incredibly difficult and slower than it would be in other languages. So for these reasons, Python was rejected as the software of choice.

2.1.2 MATLAB

MATLAB was another language that was considered. Developed in 1984, it became the goto software for many research purposes due to its ease of use and intuitive operations on matrices and its multi-dimensional analogues. It is taught across every single engineering college, and one company estimates that MATLAB is being used in over 57,811 companies [**?**]. MATLAB, however, is still an interpreted language and therefore can be quite slow some times. A study conducted by Aruoba and Fernández-Villaverde found that their MATLAB code ran about 3x slower than the same code written in C++, highlighting just how big of a difference a compiler can make [**?**]. It should be noted that the MATLAB version of the code ran about 30.26x faster than the same Python code, implying that even though both are an interpreted language, the MATLAB interpreter is much more optimized than the Python interpreter. But one of the biggest drawbacks by far, is the fact that MATLAB is a license based language. In order to use MATLAB, one must pay a yearly subscription of anywhere from €120 to €3,650 depending on the purpose for which it is used. This goes directly against the open-source vision of MERIT.jl. This cost can become prohibitively expensive for small teams trying to work on new systems and algorithms. Octave was briefly considered as it is a free and open-source competitor to MATLAB, but this idea was quickly dropped when it became clear that Octave's main goal was compatibility with MATLAB scripts rather than performance over MATLAB. For these reasons, MATLAB was also rejected as the software of choice.

2.1.3 Julia

The third and final language considered was the Julia programming language. Julia was developed by Jeff Bezanson, Stefan Karpinski and Viral B. Shah, and was first released to the public in 2012. Julia was designed to be a coding language that offered high-level syntax while also automatically compiling the code at runtime. Julia's Just In Time (JIT) compiler, allows it to offer the full dynamism of Python and MATLAB while avoiding the drawbacks that come with having to use an interpreter. The aforementioned study found that Julia was only 1.47x slower than the comparable C++ implementation. Julia also offers features that are not in Python or MATLAB such as parametric polymorphism, multiple dispatch, efficient garbage collection and a JIT that is capable of optimizations and ahead-of-time compilation. Julia also bakes in native support for multiple parallel programming paradigms such as GPU programming and multithreading, making it an appealing choice for people working on high-performance compute clusters. One other feature that made Julia wildly popular was the native ability to call C and Fortran libraries without having to create any special wrappers. This "It Just Works" ideology is what garnered Julia its well deserved popularity. With its rising fame and easy to understand syntax, this was the coding language decided for MERIT.jl. In the next few sections, the various Julia features used will be discussed in more detail.

2.2 Features in Julia

2.2.1 Multiple Dispatch

Multiple Dispatch is a programming paradigm in which a function can be dispatched (invoked) based not only on the function name but also on the type and order of input arguments. This is opposed to the single dispatch programming paradigm where the function that is dispatched usually depends on a special argument before the function name. In almost all programming languages this is the variable name for that class. For example, consider the following code:

```
Class Dog:
    name::string

    function says(a::string):
        print("The dog, $self.name says $a")
    end
end

Class Cat:
    name::string

    function says(a::string):
        print("The cat, $self.name says $a")
    end
end

billy = Dog("billy")
kate = Cat("Kate")
```

If we wanted to call the "says" function for the Dog or Cat class, then we would have to write, `billy.says("Hello World")` or `kate.says("Hello World")`. This is the single dispatch paradigm and it is probably the one most people are familiar with. This fits well into an object-oriented programming language, where objects are used to encapsulate concepts and ideas and therefore we create specialized functions that operate on the data contained within that class. However, one drawback is that the compiler relies on the user to remember which methods belong to which class, and also which methods are callable. Also, it is not clear how one would write a function that allows for the Cat and the Dog to interact with each other. Instead, now consider the multiple dispatch paradigm. In Julia particularly, it is important to note that methods no longer belong to objects as we were forced to do in single dispatch. Added to this, methods no longer have to be defined within the "class" but can be done after class declaration or even class instantiation. Consider the example below:

```
struct Dog{
    name::string
}

struct Cat{
    name::string
}
```

```
8
9     function says(pet::Cat, a::string)
10        print("The cat, $pet.name says $a")
11    end
12
13    function says(pet::Dog, a::string)
14        print("The dog, $pet.name says $a")
15    end
16
17    function encounter(petA, petB)
18        print("$petA.name encounters $petB.name and $meets(
    petA, petB)")
19    end
20
21    function meets(petA::Cat, petB::Dog)
22        return "hisses"
23    end
24
25    function meets(petA::Dog, petB::Cat)
26        return "barks"
27    end
28
29    billy = Dog("Billy")
30    kate = Cat("Kate")
31    says(billy, "Hello")
32    says(kate, "Hello")
33    encounters(kate, billy)
34    encounters(billy, kate)
```

Here we have a similar situation as before, but instead of having classes, we have structs and methods that take those structs as arguments. On lines 30 and 31 in the code block above, we get a similar output as in single dispatch. But where multiple dispatch really shines is in the next two lines. Here we have the cat and the dog encountering each other. In single dispatch, someone would have to create a wrapper library to handle the interaction between two classes, but in multiple dispatch it is as easy as creating another method. This can be seen in the "meets" functions defined in the code block above. Since multiple dispatch relies on the input arguments as well, we can create specialized "meet" functions for when a Dog meets a Cat or when a Cat meets a Dog. In the above code block executing line 32 would print out "Kate encounters Billy and hisses". This is because we had a struct of type Cat as the first input argument to "meets" and a struct of type Dog as the second argument. Line 33 on the other hand, would print "Billy meets Kate and barks", since the order of the input arguments is reversed. Another notable consequence from this is that if anyone else wanted to extend "meets" and therefore "encounters" to other animals, they would only need to define a struct with the same fields as Cat and Dog, and new meets methods as such:

```
1     struct Rabbit{
2         name::string
3     }
4
```

```
5     function meets(petA::Cat, petB:Rabbit)
6         return "chases"
7     end
8     function meets(petB::Rabbit, petB:Cat)
9         return "runs away"
10    end
11
12    kate = Cat("Kate")
13    johnny = Rabbit(Johnny)
14    encounters(kate, johnny)
15    encounter(johnny, kate)
```

Line 12 in the above code block would print "Kate encounters Johnny and chases" while line would print "Johnny encounters Kate and runs away". While these examples may be simple, they showcase the powerful flexibility and extensibility behind multiple dispatch. It is this feature that allows all the Julia libraries to "just work" together, without needing any glue-code. Some other developer could come along and create a new package for a "Lion" and so long as it implements a struct with the same required field names as the other animals and creates a "meets" function, any variable of type Lion will work well with the "encounters" function.

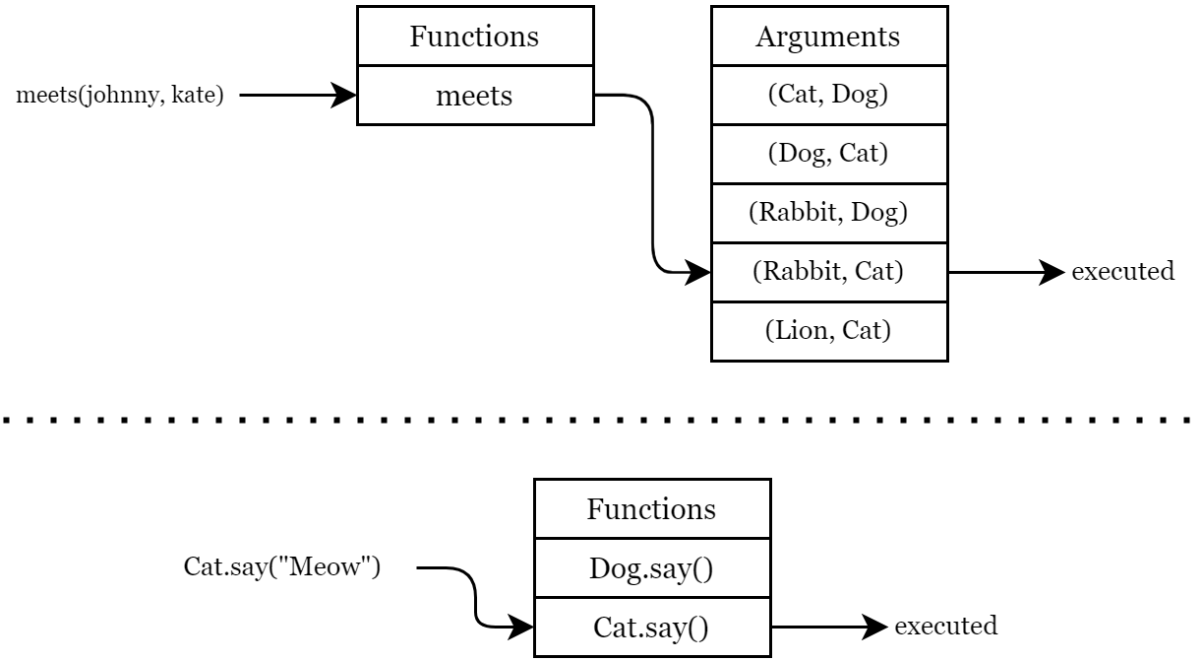


Figure 2.9: Multiple Dispatch (top) vs Single Dispatch (bottom)

2.2.2 Type Heirarchy

All the types in Julia are arranged in a tree-like structure and can be broadly classified into two categories, an Abstract Type or a Concrete Type. Abstract types are the internal nodes of the type tree, having both parents and children, while concrete types are the "leaves" of this tree. One notable difference between abstract types and concrete types is that abstract types cannot be instantiated; they serve only as nodes in the type graph. While this may seem

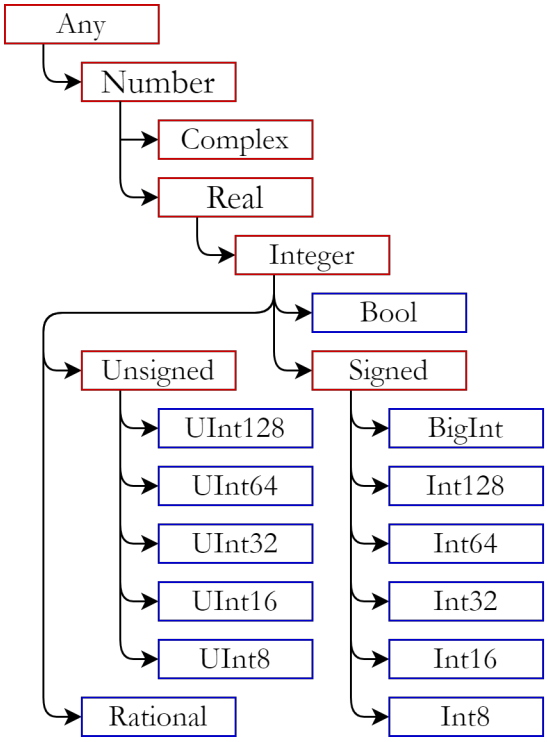


Figure 2.10: Type hierarchy for the Integer Type

pointless at first glance, it is this very feature that allows the multiple dispatch paradigm mentioned before to really shine. Shown in Figure 2.10, is the type hierarchy for the Integer type, the abstract types are highlighted in red, whereas the concrete types are highlighted in blue.

As well as giving us a way to logically organize types, the type hierarchy tightly integrates with the multiple dispatch system. Consider for example the code block below:

```
1   abstract type Animal end
2
3   struct Cat <: Animal
4       name::string
5   end
6
7   struct Dog <: Animal
8       name::string
9   end
10
11  function encounters(petA::Animal, petB::Animal)
12      verb = meets(petA, petB)
13      println("$(petA.name) meets $(petB.name) and $(verb)"
    )
14  end
15
16  meets(petA::Animal, petB::Animal) = "passes by."
17  meets(petA::Cat, petB::Dog) = "hisses"
18  meets(petA::Dog, petB::Cat) = "barks"
```

```
19
20      whiskers = Cat("Whiskers")
21      fudge = Dog("Fudge")
22      encounters(whiskers, fudge)
23
24      ##################################################
25      #Defined in another library that subtypes the Animal
    abstract class.
26      struct Rabbit <: Animal
27          name::String
28      end
29
30      chomper = Rabbit("Chomper")
31      encounters(whiskers, chomper)
```

In the above code block, we define an abstract Animal type, by default this is a subtype of the abstract type Any. Then we define a concrete type Cat and a concrete type Dog. When line 22 gets executed, the multiple dispatch system will choose the correct `meets()` function and "Whiskers meets Fudge and hisses", this is as expected. Now another developer might want to implement a Rabbit type. So they create the Rabbit type as shown above, and specifies that it is a subtype of the Animal abstract type via the `<:` operator. Even if they do not implement a `meets()` that is specialized for their Rabbit class, the `encoutners()` will still work since it only ex:wantpects an input that is a subtype of Animal. The `meets()` function will also work, since the original author implemented a `meets()` method that accepts arguments of type Animal or any subtypes of it. But why did not the earlier call to `encoutners()` (and consequently `meets()`) fail, since both lines 15 and 16 are valid options for the dispatcher? This is an example of where multiple dispatch benefits from having a type hierarchy. The Julia dispatcher always dispatches the function that is most specific across **ALL** its arguments. So in the Cat and Dog case, the `meets()` on line 16 will be dispatched. In the Cat and Rabbit case, the `meets()` one line 15 is the most specific function and so that gets dispatched. This allows for amazing extensibility that isn't found in many other languages. Often it is impractical for a developer to think of all the ways their library would be used or all the types or functions that the end-users would require. In Julia, they don't have to think about it. Instead, they can define an abstract class and some methods that accept the abstract class as inputs. The users can then create subtypes of this abstract class and define their own methods for their new type. Neither developer has to be worried about whether they were through enough to consider all possibilities, they can write code for their own purposes and trust that the Julia dispatcher and type hierarchy would automatically determine the correct function to call.

This idea is heavily used in the MERIT.jl library. Currently, most of the research in microwave imaging is centered around the breast and breast cancer, but in the future, this could be implemented for other body parts. As such, the library needs to be flexible to allow for easy updates. MERIT.jl is centered around the Scan abstract type, from this type we subtype the BreastScan type, which holds all the information from the scan of a particular breast. If in the future microwave imaging gets used for a chest scan, all a developer would need to do is to implement a ChestScan struct, subtyped from Scan, and implement a few functions and the Julia dispatcher will handle the rest. I, as the original developer, do not need to worry about the intricacies of how a ChestScan struct would work, or what fields it would need. I can trust

that future developers can easily extend this library without introducing breaking changes to the library core. In this way, MERIT.jl achieves extensive flexibility and expandability which would not be possible in other languages.

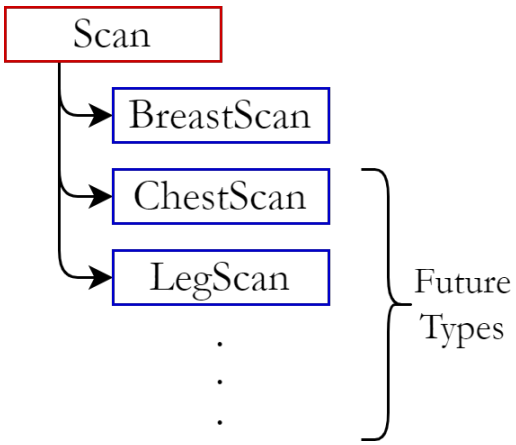


Figure 2.11: Current type hierarchy in MERIT.jl

2.2.3 Parametric Polymorphism

Parametric polymorphism refers to the programming paradigm where a function can be created generic to the input argument type. The compiler can then strongly type the function with the correct types. Consider the simple example of an `addtwo(x)` function.

```
function addTwo(x::T) where {T <: Real}
    return x + T(2)
end
```

Instead of restricting x to a specific type, we replace it with the variable T and instead place a restriction on T. In this case we say that T, and by extension x, can be any type that is a subtype of the abstract Real type. We also convert the 2 from an Int64 type, to whatever type T is, to allow for accurate summation. If parametric polymorphism was not a feature in Julia, this function would need to be duplicated 16 different times to account for all the concrete subtypes of Real (see Figure 2.10). Parametric Polymorphism with the Julia type hierarchy allows a developer to have incredible expressive power with very few lines of code. MERIT.jl uses this feature extensively due to the fact that the library needs to be agnostic to the type of the input data. When instantiating the BreastScan struct the user can provide the types of the data that they will load and can therefore set the internal data type of the struct. It would be impossible to support all 4,176 different permutations of every single function, however, with parametric polymorphism, the function only needs to be defined once and the Julia compiler will handle the rest. This means that researchers and developers don't have to worry about whether the library functions can support the data type of their data, so long as it follows the type restriction set on the function, it will produce an output, greatly simplifying the coding experience.

2.2.4 Type Stability

Type Stability is a coding discipline that Julia recommends all code to use. A function whose contained variables have a consistent type for the lifetime of that function is considered to be "type stable" or to have "type stability". Another way to think about this is that if the output type only depends on the input types, then the function is type stable. Consider the following two functions:

```
function unstable()
    x = 1
    for i = 1:10
        x /= rand()
    end
    return x
end

function stable()
    x = 1.0
    for i = 1:10
        x /= rand()
    end
    return x
end
```

In both functions, we have a variable x which gets divided by a random float 10 times, with the result being returned. However, the first function is considered to be type unstable while the second function is type stable. This is because in the first function x was declared as an Int64, but when we divide by the random Float64, the data type of x changes to Float64 as Julia performs floating point division by default. Whereas in the second function, x gets declared as a Float64 and remains as a Float64 when it gets returned. Type Stability is important because if the Julia compiler can determine that the types of the variables stay constant for the lifetime of the function, it can perform optimizations on the machine code that would otherwise be impossible. These optimizations are evident when benchmarking the functions above. Using the `@benchmark` macro from BenchmarkTools, I timed both functions. They were run multiple times to ensure that only the compiled versions of the function were being executed rather than the functions being interpreted. The benchmarks showed that on average the type stable function ran in 306.178ns whereas the type unstable function ran in 573.355ns; almost 1.87x slower. When considering the many hundreds of function calls required to produce an output in MERIT.jl, these slowdowns become significant. When coding functions for MERIT.jl, I aimed to preserve type stability in functions that I knew would be called frequently. The `@code_warntype` macro was greatly beneficial in this task as it flags sections of code where the compiler cannot definitively infer the type of a variable or where it suspects there might be some type instability. Having type stable functions in MERIT.jl was a necessity, where possible, due to the large datasets that need to be processed. An 8cm radius breast at 0.25cm resolution has about 70k points, which need to have their distances to each antenna calculated. Assuming we are dealing with the MARIA M4 system with its 60 antennas, this equates to around 4.2M distance calculations alone for each image. Running such calculations on type unstable functions would be prohibitively expensive and would take far too long to compute. Herein lies one of the drawbacks of using the Julia language. While

the dynamism offers us powerful expressibility, it also comes at the cost of massive slowdowns when used incorrectly. While it may be easy to create type stable functions for the example shown in the code block above, it becomes increasingly harder to create type stable code when the operations that need to be performed become more complex. Creating performant code in these situations requires the developer to have advanced knowledge of the language. This does raise the barrier of entry for people who want to contribute to the library. However, I feel that this is an acceptable trade-off since the benefits that come from having a performant library, such as increased usage and better recognition, far outweigh the negatives from having an increased barrier of entry.

2.2.5 Closure

Closure in programming refers to the practice of calling a function A which returns a function B that has some information about the variables in the scope of function A. Consider the following simple example:

```
1    function addX(x)
2        scalar = x
3
4        function calc_(a)
5            return a + scalar
6        end
7    end
8
9    add5 = addX(5)
10   add7 = addX(7)
11   add5(2)  # Will return 7
12   add7(10) # Will return 17
```

Here a function called `addX` is defined which accepts an input `x`, assigns it to a variable called `scalar` and finally defines a function called `calc_` which calculates and returns `a + scalar`. Here we say that `scalar` is "captured" by `calc_`. When calling the `addX` function as was done in line 8, we receive a function pointer of sorts to a parametrized `calc_`. Note that by default Julia automatically returns the last thing computed in the executed function's scope, this is how `calc_` is returned from `addX` without an explicit return statement. This idea of closure is important as it allows users to customize parametric functions to suit their particular needs. The MERIT.jl library uses this in the `get_delays` function in the Beamform.jl file (for more information see the GitHub link in the Introduction). Earlier sections showed how *epsilon* is a free parameter in the beamforming equations, one that researchers would be changing frequently. Implementing closure for the `get_delays` method allows researchers to quickly and easily define a set of delay functions which they can pass to the `beamform` function to see the various effects on the resulting scan that come from tweaking *epsilon*. This fulfills one of the other tenets of the library, which was to create an interface that allowed for flexibility and customizability in the functions that mattered. One important thing to keep in mind is that captured variables must never be reassigned otherwise, the compiler will not be able to infer the data type of the variables leading to slow and type unstable code. Again this highlights one of the drawbacks of using Julia, writing customizable code is relatively easy, however writing code that is both customizable and performant is tough. Any researcher who might want to implement a custom delay function (e.g. one that does not assume a straight line

model) using closure, would have to make frequent use of the `@code_warntype` to ensure that the code they have written is type stable, and therefore comes with some promises of performance.

2.2.6 Type Safety

Type Safety refers to a program or language's ability to detect and discourage errors that arise from performing operations on the wrong data type. For example in Julia, one can easily add two numeric types together but if one tried to add two strings together, the Julia compiler would produce an error, since that would be an illegal operation. This offers some level of protection against illegal operations, but there are many cases where the type of the variable agrees with the operation being performed, but their semantic meaning disagrees. Consider the example of a function used to calculate the speeds of various vehicles. The function `calcSpeed` accepts a vector for distance traveled by the various cars and the times each car took to travel that distance and returns the speed of each car in a third vector.

```
1  function calcSpeed(dist::Vector{T}, t::Vector{T}) where {T <:
       Real}
2      return dist ./ t
3  end
4
5  ######################################################
6  distance = rand(1, 100)
7  time = rand(1, 100)
8  calcSpeed(distance, time)    # Will compute the speed
9
10 ######################################################
11 distance = rand(1, 100)
12 time = rand(ComplexF64,1, 100)
13
14 # Will throw an error since there is a type mismatch
15 calcSpeed(distance, time)
16
17 ######################################################
18 distance = rand(1, 100)
19 time = rand(1, 100)
20
21 # Will compute the wrong answer, but no error gets thrown
22 calcSpeed(time, distance)
```

On line 8, calcSpeed gets called correctly and returns the speed of each car correctly. On line 13, the Julia compiler will throw an error because the `time` variable is a Complex Float64 and `calcSpeed` only accepts types that are a subset of Real. However, on line 18, no error is thrown even though the wrong answer is returned. This is because compilers can only make deductions on the concrete types of the variables and the operations being performed and not on the semantic meaning of the function, variable names and what it means to call the function with those variables. However, we can limit the chance of a variable being passed to the wrong positional argument by creating lightweight types that encode this semantic information in their type name. In the above case, it would mean creating a "distance" type,

and a "time" type that are wrappers around some in-built concrete type. With this, the compiler will throw an error when running line 18 as the function received a vector of type "time" when it was expecting one of type "distance".
The MERIT.jl library exemplifies the idea of "strong" type safety through its implementation of the Point data type. The Point type is an abstract type from which the Point3 and Point2 concrete type subsets. These are lightweight wrappers around a grouping of 3 and 2 numbers respectively and serve the purpose of being a 3D and 2D point.

```
1   abstract type Point end
2
3   # xyz can be any data type that is a subset of Real
4   mutable struct Point3{T <: Real} <: Point
5       x::T
6       y::T
7       z::T
8   end
9
10  mutable struct Point2{T <:  Real} <: Point
11      x::T
12      y::T
13  end
```

Since these are custom data types, the inbuilt operators could not be used on them. So in addition to this, MERIT.jl also had to extend some of the inbuilt operators using the concepts from parametric polymorphism so that these points data types could be used in a meaningful way. For the full suite of operations implemented, please refer to the GitHub link in the Introduction. Every other function in the library that needs to work with the points from the imaging domain, e.g. the `domain_hemisphere!` and the `get_delays`, to name a few, accept collections of the points data types rather than simply accepting a matrix of numbers. This way, an error gets thrown if any other collection of numbers that is not a points data type is passed in that argument. This does not entirely solve the problem however, it is still feasible for a Point3 variable representing antenna locations to be mistakenly used in the place of a Point3 variable which describes the imaging domain. Even though this issue still exists in some part, the hope is that with this new type, the chance of the aforementioned issues arising would be minimal. Due to time constraints, the entire library could not be made strongly type safe, but the Points.jl file provides an excellent template that could be used to implement further types to eventually reach a strong type safety that is in every function in the library.

2.2.7 Customizability

The customizability of the MERIT.jl library has been demonstrated extensively in the previous sections. However, nowhere is this exemplified more than in the BreastScan struct. The entire library is built around the use of the Scan structs. These structs encapsulate all the information about a particular Scan, including all the required information about the machine that was used to conduct the scan. Most importantly each scan struct is recommended to have two function pointer fields which will hold pointers to the delay function and the beamforming function to be used in the beamforming process. During the setup process, researchers can easily swap out the delay function or the beamforming function used by overwriting these

function pointers with other relevant functions from the library or with their own functions. In order to work with the one-call data processing pipeline in MERIT.jl, researchers just need to ensure that their delay and beamformer functions follow these templates:

```
function delay_template(relative_permiativity)
    #Capture the relatively_permiativity
        function(channels, antenna, domain_points):
            #calculate and return a time matrix
            #Size = (1 x #Channels x #Points)
        end
end

function beamformer(delayed_signals)
    #Do some processing
    #return should be of size (1 x 1 x #Points)
end
```

These templates allow for a lot of flexibility and are really where MERIT.jl shines in comparison to its MATLAB counter part. Researchers who just want to benchmark new algorithms against already established ones just need to follow these templates and they can be guaranteed that their functions will "just work" with the one-call data processing pipeline. If their functions also follow the rules of type stability mentioned above, they can also have some reasonable guarantees that any slowdowns are caused by their implementations rather than the lack of compiler optimizations for type unstable code.

Results

3.1 Current Workflow

The current workflow in MERIT.jl was designed to be simple and approachable without much background knowledge about how Julia or the underlying library works. Shown below is the full workflow needed to generate a plot from the data:

```
using MERIT
using Plots


plotlyjs()
scan = BreastScan{Float32, ComplexF32, UInt32}()
domain_hemisphere!(scan, 2.5e-3, 7e-2+5e-3)
load_scans!(scan,"data/B0_P3_p000.csv" , "data/B0_P3_p036.csv
   ", ',')
load_frequencies!(scan, "data/frequencies.csv", ',')
load_antennas!(scan, "data/antenna_locations.csv", ',')
load_channels!(scan, "data/channel_names.csv", ',')
scan.delayFunc = get_delays(Float32(8.0))
scan.beamformerFunc = DAS
image = abs.(beamform(scan))
imageSlice = get_slice(image, scan, 35e-3)
graphHandle = heatmap(scan.axes[1], scan.axes[2], imageSlice,
    colorscale="Viridis")
savefig(graphHandle, "GettingStarted.png")
```

A user would first instantiate the BrestScan struct and assign the data types that would be used throughout the processing pipeline. The first type sets the data type and thereby the precision of the points composing the imaging domain, the antenna locations and the frequency divisions. This can be any datatype that is a subset of type Real. The second type controls the data type of the signal matrix containing the data collected from each antenna. This can be any type that is a subtype of Number allowing for time-domain (Real) signals or frequency-domain (Complex) signals. The third type controls the data type of the channels, and it can be any type that is a subtype of Integer. The choice of data type here has no accuracy impact on the final result and it is recommended to choose an Unsigned Integer data type that is big enough to index all the antennas. The next step would be to generate the imaging domain, this is accomplished using the `domain_hemisphere!` function. This accepts the resolution and the assumed or calculated radius of the breast. The user would then have

to make use of the `load_XXX!` functions to load the data into the relevant fields of the struct. These functions assume the data is contained in a CSV file and contain no headers. The user then populates the relevant fields with their chosen beamformer and delay function. At this stage, the user can then pass the whole struct to the `beamform` which will beamformer the provided signals into a set of data that can then be visualized as demonstrated towards the end of the code block above. Overall the entire workflow can be seen in Figure 3.12.

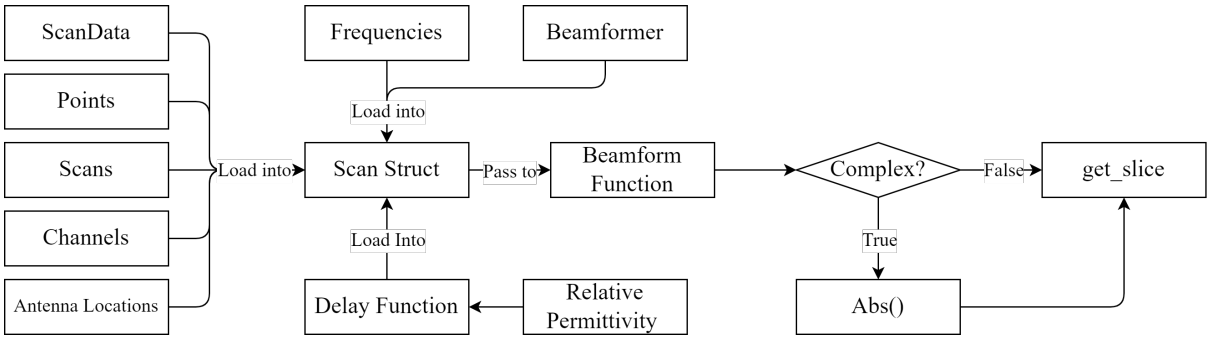
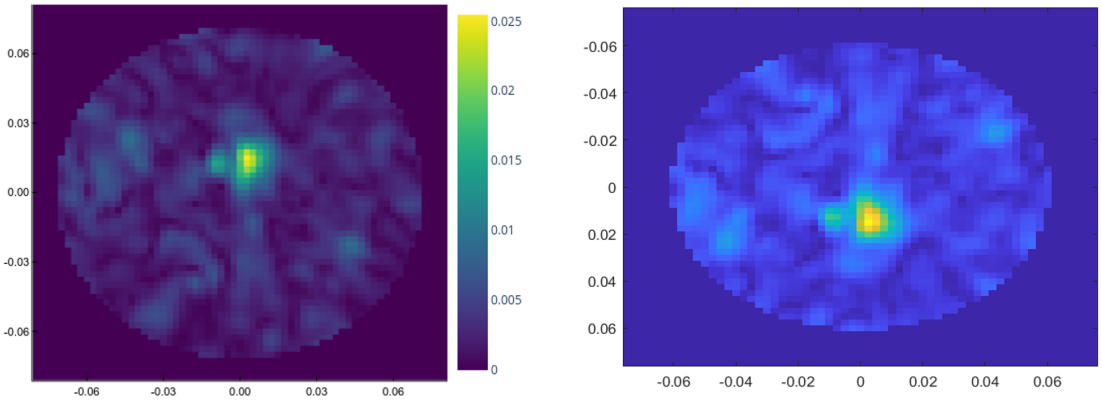


Figure 3.12: The entire MERIT.jl Workflow

This workflow exemplifies how easy it is to use the MERIT.jl library. The function names were deliberately chosen to be verbose and conform to established nomenclature in the field so that anyone who wants to use the library can understand what each function does without having to delve into the source code. In this way, the MERIT.jl library is somewhat self documenting. Additional information is provided above each function in the form of docstrings which can be used with the help function built into the Julia REPL. This was done in a bid to lower the prerequisite knowledge needed to use the library, thereby fulfilling one of the goals of MERIT.jl which was to create an open and acessible library.

3.1.1 MERIT.jl Results

The library was benchmarked against the MATLAB implementation created by Prof O'Loughlin et al to ensure that the results provided by MERIT.jl are provably correct. Both implementations were given the same data, B0_P3_p000.csv and B0_P3_p036.csv. Rotational subtraction was performed on these in both libraries to reduce the presence of skin reflections in the data as mentioned before. The data was then processed according to the processing pipeline recommended by both libraries, the result of which can be seen below in Figure **??**. It should be noted that the `imshow` function was used in MATLAB to plot the image. This had the effect of reflecting the image across the x-axis and also slightly stretching the image along the x-axis, however, the matrix holding the image data still shares the same layout as the image matrix in Julia so a numerical comparison between the two could be performed. The averaged MSE proved to be an excellent choice as a numerical comparator, due to the squaring operations in the MSE formula any small differences would be greatly magnified in the error. This is desirable when the goal is to see if MERIT.jl can provide output that is similar to its MATLAB counterpart. Computing the averaged MSE between the two images yielded an error of 8.4417×10^{-7} which is well within the accuracy of a float, making it effectively identical to the images produced by MATLAB. This shows that the Julia library in its current state provides a viable alternative to the MATLAB implementation for frequency domain analysis.

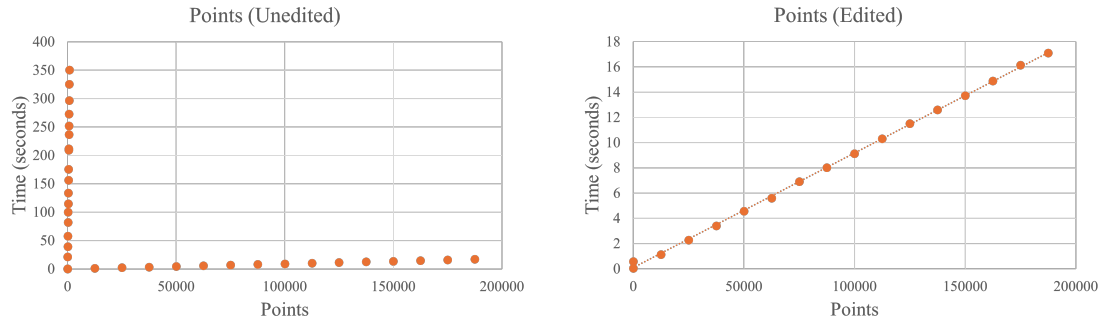


(a) MERIT.jl Output

(b) MATLAB Output

3.1.2 Performance of MERIT.jl

One of the requirements for MERIT.jl was that it had to be performant. Through the use of type stability, SIMD optimized for loops and the disabling of array bounds checking, the library could process the provided data in 8 seconds. However, after the addition of the Points data type for type safety, the processing time climbed to 12 seconds. While the overall runtime is still acceptable, an increase of 3 seconds is less than ideal. While no official analysis has been conducted to narrow down the cause of the slowdowns, I suspect that perhaps it may be down to unoptimized implementations of the basic operators such as addition and squaring in the Points.jl library. It should also be noted that when running the library, my laptop constantly ran into memory limits and had to write some memory to a swap file on disk. I believe that moving memory in and out of this file might also be partly responsible for the increased runtime, however, further research is necessary to determine a definitive cause. To capture the full performance characteristics of the library, a scalability test was performed, in which the number of points, channels and frequency divisions were progressively increased. This was performed in an automated manner using the functions provided by BenchmarkTools.jl [?]. The results from the benchmark suites were then exported to CSV files and analyzed in Excel to judge the "Big O" notation of MERIT.jl.



(a) Unedited Points

(b) Edited Points

Figure 3.14: Runtime for increasing Points

Shown above in Figures 3.14, 3.15, 3.16 are the results from BenchmarkTools. The reason for the two types of graphs was to remove outliers that skewed the data and prohibited any analysis. Running the benchmark used all the free RAM available on my laptop as well as

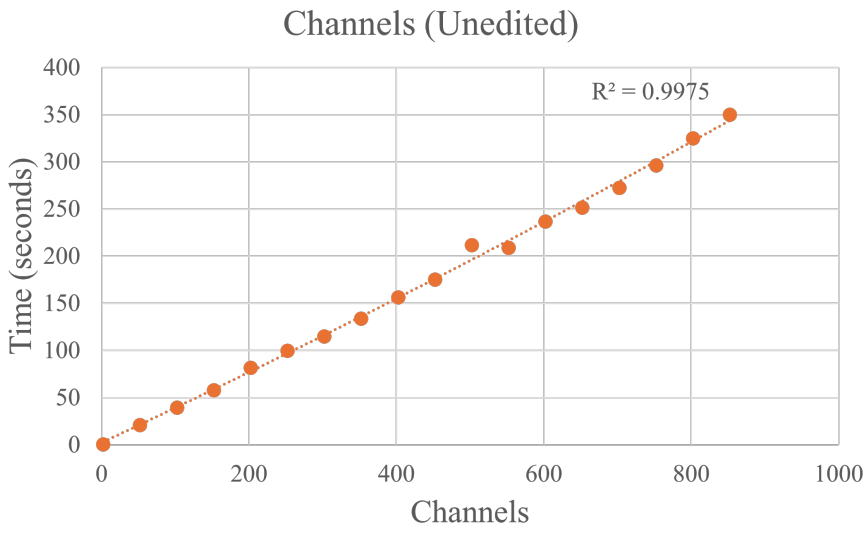
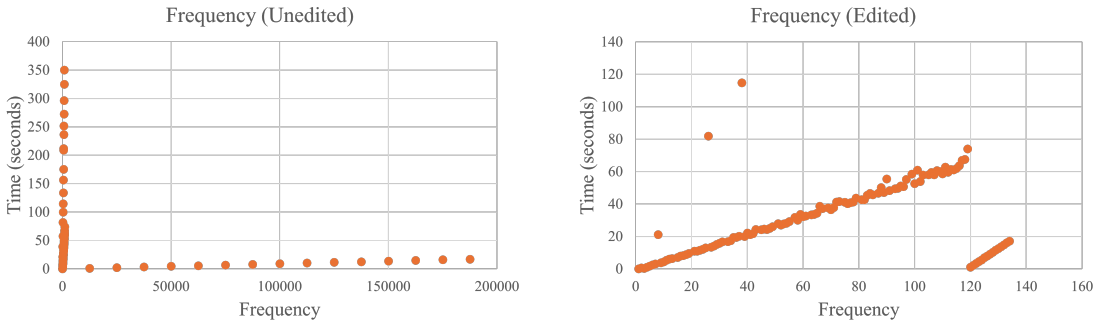


Figure 3.15: Runtime for increasing Channels



(a) Unedited Frequencies

(b) Edited Frequencies

Figure 3.16: Runtime for increasing Frequency divisions

significant amounts of swap space on my drive due to large amounts of overhead introduced by the BenchmarkTools benchmark. I believe that the latency caused by IO to the swap file was the cause for the outliers and so I removed them in the edited graphs since these points are not indicative of the actual performance of the library. After these points were removed, a curve was fitted to each graph to provide an indication of the time complexity. From this analysis, I concluded that the runtime of the library scales linearly ($\mathcal{O}(n)$) with the number of points, and quadratically ($\mathcal{O}(n^2)$) with the number of channels. This aligns with expectations based on the code analysis of the DAS implementation. The main beamformer loop consists of a single for loop iterating over the points as such, the algorithm is expected to behave linearly with an increase in points. With channels, every additional antenna corresponds to a quadratic increase in channels since every antenna can send to and receive from any other antenna in a fully multistatic system. This is evidenced by the slight quadratic increase in Figure 3.15. The algorithm is expected to have $\mathcal{O}(n)$ growth since the frequency is only used once in the algorithm process to delay each signal, which is evidenced by the linear growth in 3.16. Effectively, MERIT works very well with a large number of points and with relatively few channels. This in turn affected the layout of the matrices in the library. Since data to do with points would be accessed most frequently, the decision was made to place these along

the columns of the matrices. Since Julia is a column major language, this would provide the quickest access to this data. Data that relate to the channels were placed along the second dimension since this would be accessed less frequently than data related to the points. Finally, data relating to the frequency were placed along the third dimension, since this data is accessed very infrequently, it was decided that the latency required to load this data from memory would be acceptable provided it allowed us easy access to data relating to points and channels. The above graphs demonstrate the performance of the MERIT.jl library. From limited testing on my laptop with an Intel i7-1185G7 CPU and 16GB of RAM, the Julia library executed in the same amount of time as its MATLAB counterpart. However, further testing to accurately quantify the runtime of both libraries is needed.

Future Work

4.1 Time Domain Implimentation

Due to the limited time I had to work on this library, I mainly focused on implementations of beamformers in the frequency domain. While this covers some systems, the library effectively excludes a whole class of systems that perform time domain data gathering. But I don't see this being difficult to include in future updates. Using the multiple dispatch feature in Julia, one could extend the `delay_signal!` in the Process.jl file to accept signals that are a subtype of Real instead of Complex. The beamformers can stay largely the same since they do not depend on the type of input data. The rest of the pipeline should work with the time domain since they are type agnostic, or they have a type restriction permissive enough to allow for both time and frequency domain signals.

4.2 Implimentation of More Beamformers

Currently, the only beamformer implemented is the DAS beamformer, again mainly due to time constraints on the project itself. However, in the future one could consider implementing a generalized DAS beamformer, where more than just the relative permittivity value is parametrized. A more generalized DAS beamformer can described mathematically as:

$$I_{\epsilon_i}(r) = \mathcal{G}(\mathcal{S}) \left[\sum_{\Omega} \sum_{\mathcal{A}'} \sum_{\mathcal{A}} S_{a,a'}[\omega] e^{j\omega \tau_{\epsilon_i,a,a'}(r)} \right]^2 \tag{4.6}$$

Here $\mathcal{G}(\mathcal{S})$ can be considered as a generalized weighting function that can be defined by the user. This way, the library needs only to support this one family of beamformers which can then be specialized into traditional DAS or even Weighted DAS. Programmatically, this could be implemented via closure the same way `delay` was defined in Beamformer.jl. It would also require a rewrite of the one-call processing pipeline that has been established via the `beamform` function. However, this is expected since the one-call function is only meant for researchers who quickly want to visualize a scan and investigate the effects of tuning a limited set of parameters. It is assumed that anyone who would be willing to create their own beamforming function would create their own processing pipeline using the high-level functions provided.

4.3 Parallel Processing

Parallel processing was a feature that was not explored as it was outside the scope of this bachelor's project. However, there are areas of the code that have been identified as "embarrassingly parallel". These are sections of the code tha are amenable to significant acceleration through the use of multithreading and parallel processing. Consider for example the beamformer implementations, in these equations the response at each point is calculated independently of all the other points. As such this operation can be easily split across all available threads or even all available GPU cores, providing exponential increases to the performance of the library overall. The Julia language provides native support for threaded for-loops through the use of the `Threads.@threads` macro, which will evenly split the for-loop range across the threads available to the Julia runtime. However, the onus still lies on the user to ensure that no data race conditions can occur. Julia also supports GPU programming natively through the use of CUDA.jl for Nvidia GPUs, AMDGPU.jl for AMD GPUs, oneAPI.jl for Intel GPUs as well as Metal.jl for the current Apple integrated GPUs [**?**]. Out of the APIs listed, CUDA.jl is by far the most advanced and complete library due to its age and dominance in other fields and would probably offer the most benefit for researchers as they most likely already have access to an Nvidia GPU.

Bibliography

[1] A. W. Preece, I. Craddock, M. Shere, L. Jones, and H. L. Winton, "MARIA M4: Clinical evaluation of a prototype ultrawideband radar scanner for breast cancer detection." vol. 3, no. 3, p. 033502.

[2] B. M. Moloney, P. F. McAnena, S. M. Elwahab, A. Fasoula, L. Duchesne, J. D. Gil Cano, C. Glynn, A. O'Connell, R. Ennis, A. J. Lowery, and M. J. Kerin, "The Wavelia Microwave Breast Imaging system-tumour discriminating features and their clinical usefulness." vol. 94, no. 1128, p. 20210907.

[3] J. Bourqui, J. Sill, and E. Fear, "A Prototype System for Measuring Microwave Frequency Reflections from the Breast," vol. 2012, p. 851234.

[4] E. C. Fear, J. Bourqui, C. Curtis, D. Mew, B. Docktor, and C. Romano, "Microwave Breast Imaging With a Monostatic Radar-Based System: A Study of Application to Patients," vol. 61, no. 5, pp. 2119–2128.

[5] G. Stoet, PsyToolkit Testimonials, Std. [Online]. Available: https://www.psytoolkit.org/#_testimonials

[6] M. Klemm, I. Craddock, J. Leendertz, A. Preece, and R. Benjamin, "Improved Delay-and-Sum Beamforming Algorithm for Breast Cancer Detection," vol. 2008.

[7] B. Maklad, C. Curtis, E. Fear, and G. Messier, "Neighborhood-Based Algorithm to Facilitate the Reduction of Skin Reflections in Radar-Based Microwave Imaging," vol. 39, pp. 115–139.