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 12, 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 12, 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

Microwave imaging has seen a rising interest in the medical field evidenced by the numerous clinical trials that 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 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 provides researchers with all the tools needed to easily test different algorithms; the MERIT toolbox developed by Prof O'Loughlin, M. A. Elahi, E. Porter, et. al [5]. Other fields of research have seen numerous benefits from the introduction of comprehensive open-source libraries. Libraries such as PsychoPy and PsyToolkit have allowed psychology researchers to design and conduct experiments in a matter of hours by packaging common functions in an easy-to-use library [6]. With a 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

The rest of the report will be divided up as follows:

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

MERIT.jl being an open-source library has all its code available on GitHub for anyone to view and amend. It can be viewed at the following URL: `https://github.com/AaronDinesh/MERIT.jl`

Background

1.1 Literature Review

This section reviews the current state-of-the-art hardware in the field. The field has seen intense research since E. Larsen and J. H. Jacobi released a paper on microwave imaging of an isolated canine kidney, showing that this new modality provided a viable alternative to established methods such as X-rays [?]. In recent years we have seen many clinical trials for imaging systems with various competing configurations and no clear benefits in choosing one configuration over another. Despite the lack of a prevailing paradigm, some patterns can be gleaned from these trials that are broadly representative of the direction that the field is moving towards. 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. Fully Multistatic
2. Leveled Multistatic
3. Monostatic

These terms relate to the position of antennas around the breast tissue and how the resulting scan data would be structured. The subsequent sections will consider an example of each of the above configurations.

1.1.1 MARIA M4

The first system to be reviewed 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 configuration of 16 UWB antennas to the current system which is equipped with 60 antennas. These all operate in a multistatic configuration, allowing any antenna in the array to transmit to and receive from any other antenna in the array, an example of which can be seen in Figure 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

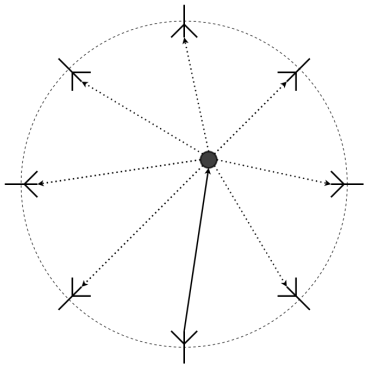


Figure 1: Example of a Fully Multistatic Configuration (Top-Down)

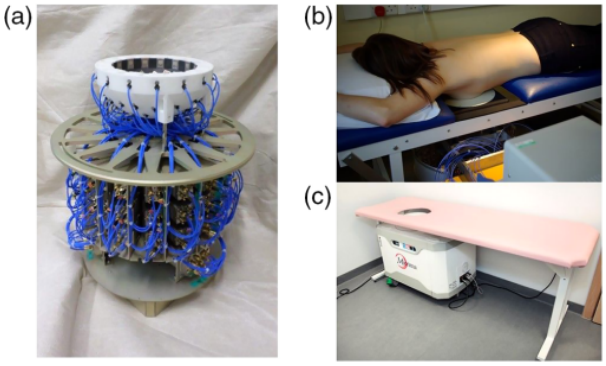


Figure 2: The MARIA M4 and M5 system. (a) The MARIA M4 antenna array. (b) The M4 in a clinical setting. (c) The integrated M5 package [1]

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 was of higher importance than price. While not explicitly stated, one would presume that the M5 system would replace the VNA with some application specific 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 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 ultrasounds and mammograms (these scans being the control)
- Be able to lie prone
- Have cup sizes between 310 and 850 ml.

The types of lesions included in the study were mostly cysts and cancers, but some other

Figure 3: The Wavelia System [2]

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 or mammogram 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, discovering sensitivities of 75% and 73% respectively. However, the credibility of these results are questioned when considering the limited 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 M4 system is a viable alternative to mammograms, it is enough to show promise.

1.1.2 Wavelia

The second system 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 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 prone on the examination table and place their breasts in the circular cutouts on the bed. Through the use of a stereoscopic camera, a 3D scan of the breast is collected allowing for an explicit calculation of the imaging domain, rather than the hemispherical approximation approach that needs to be taken with the MARIA system. The Wavelia system also makes use of the UWB spectrum while imaging, but opts to use a narrower part of the spectrum, 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

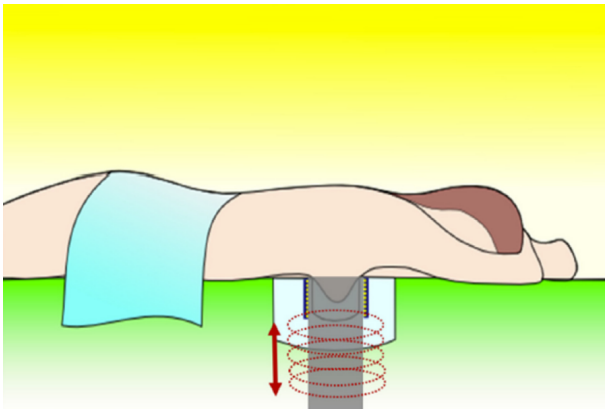


Figure 4: The Leveled Multistatic Approach of Wavelia [2]

(Figure 4). This is a leveled multistatic system as opposed to the fully multistatic system in the MARIA M4. This approach has the benefit of a theoretically infinite vertical resolution. If the radiologist requires a finer resolution along the coronal plane, they would simply change the vertical step size of the array, rather than having to manufacture an entirely new antenna array, like in the MERIT system. This leveled approach allows for the parameters of the reconstruction algorithm to be tuned independently for each coronal slice. Whereas in the fully multistatic approach, significant logic would be required in the post-processing steps to determine which channels are coplanar with a particular slice.

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:

- Have a mammogram that was performed at the time of symptomatic presentation to the breast unit (these were used as controls)
- Be capable of assuming the prone position for a length of 15 minutes
- Have a bra size larger than a 32B and have a breast size equivalent to a B cup
- Have a breast such that when submerged, there would be a margin between the cylindrical container and the breast to accommodate the transition liquid

The suitability of the patient based on the final criterion was determined by a 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 later being 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 that it cannot detect any lesions smaller than 10mm. This is significant since the size of the detected lesion plays a big factor when deciding whether it 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 prone position assumed by the patients, their breast tissue needs to have a pendulus reach far enough such that multiple sections of the breast can 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 aforementioned systems. In the MARIA and Wavelia systems, reflections from the skin can dominate in the received signals leading to artifacts in the final image. To combat this, both systems record an additional scan, where the antenna array is offset by a fixed rotational amount in the coronal plane. Any skin reflections that appear in the first scan would also appear in the subsequent scan with a similar intensity and timing, while the signals reflected from the tumor would appear at a different time position, provided that the tumor does not lie on the axis of rotation. The method aptly known as "Rotational Subtraction" involves subtracting the scans from each other, suppressing the skin reflections while preserving the tumor response. However, as noted by H. Benchakroun and Prof O'Loughlin in their papers, Rotational Subtraction can introduce artifacts into the signal due to differences between the original and rotated antenna positions relative to the tumor. These artifacts tend to appear as "wave-like" inclusions on the generated image. Both studies also noted the presence of an "echo" in the final image that appears as a duplication of the tumor response close to the true location of the tumor. They observed that the degree of rotation has an impact on the relative amplitude of the response and its echo citing that for tumors greater than $3cm$ away from the center, an increased rotation angle ($> 20^{\circ}$) caused the tumor to go in and out of focus. For rotations less than 15° the tumor could not be identified in the image [**?**, **?**]. The TSAR system, on the other hand, makes use of an adaptive algorithm that estimates the skin response at an antenna as the weighted sum of 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 submerged in an immersion liquid. However, unlike the previous two systems, TSAR operates in a monostatic configuration. In this setup, there is only one antenna that acts as both the transmitter and receiver. This antenna, usually fixed on some type of rotating apparatus, would move 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 commercial-grade VNA. To help with the reconstruction of the imaging domain during the post-processing step, TSAR makes use of a laser that explicitly 3D scans the breast. The prototype setup can be seen in Figure 5

One feature of a monostatic configuration is that the number of antenna locations per row and the number of rows are parameters that can be changed by a radiologist to balance the trade-off between speed and precision. The clinical trial conducted by this study was extremely limited, only including 8 successfully imaged patients. Due to the low sample size, one cannot reliably say whether this system and its adaptive reflection-suppressing algorithm can outperform the other two systems in terms of accuracy.

Figure 5: The TSAR Prototype [3]

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 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 6.

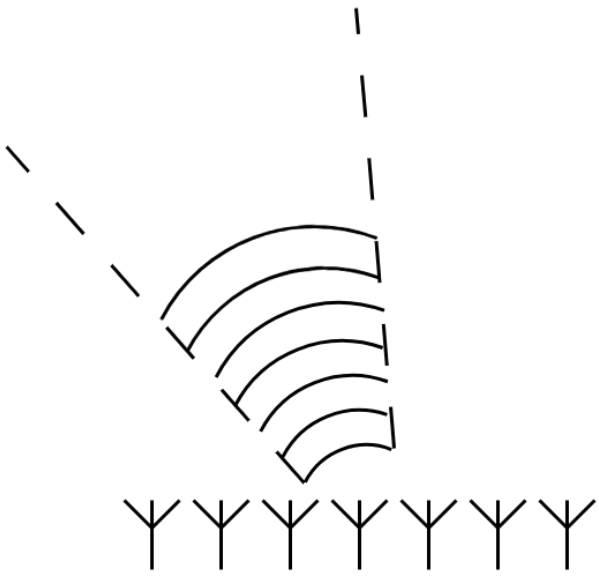


Figure 6: Phased Array Diagram

This is the forward beamforming process, but one can also consider the reverse process. Considering Figure 7 imagine a point emitter embedded in a 2D plane, that radiates circular waves evenly in all directions, with the waves falling incident on an antenna array. Due to

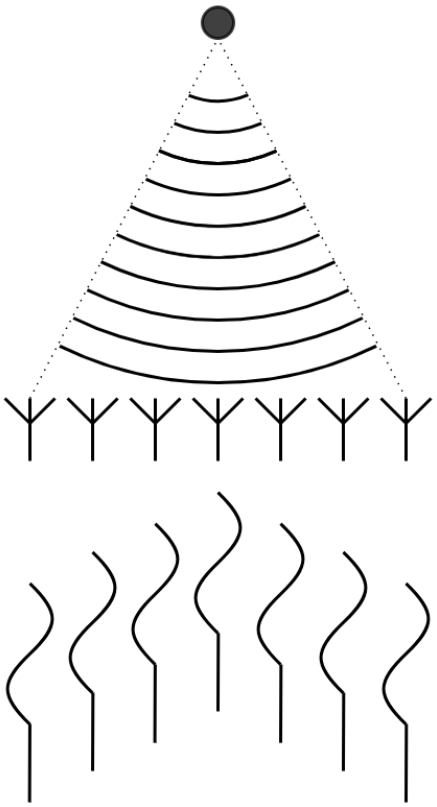


Figure 7: Point Emitter with a Phased Array

the diffusion of the wave through space, the same wavefront will appear at each antenna at different times. This manifests in the recorded signals as similar amplitudes but shifted in time. 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 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 will work on this assumption, but these equations can easily be converted to the relative time domain functions, by replacing the multiplication by a complex exponential with the equivalent time delay. As stated before a point from the imaging domain is chosen (r). The path delay of the wave from A to $\mathcal{A}'$ via r is estimated via the following equation:

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

with ϵ being the relative permittivity of the medium. In reality, ϵ changes from patient to patient and even varies within the breast of each patient depending on the path taken. However, to achieve a practical beamformer, one must estimate an averaged relative permittivity value for the entire breast, henceforth this will be labeled as ϵ_i and it will parametrize the DAS beamformer. Equation 1.1 also assumes a straight-line path between r, $\mathcal{A}$ and $\mathcal{A}'$, even though in most cases, this assumption does not hold. However, as found by Prof O'Loughlin, B. L. Oliveria and M. A. Elahi et al, this assumption yields a maximal error of $3mm$ in position, while greatly simplifying the delay calculations [8]. Using this, the received signals are delayed, summed across all stepped input frequencies and finally squared to yield the energy at the chosen point. 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 implemented as the beamformer of choice due to its simplicity and time constraints placed on the project. Two other well known beamformers will be discussed for completeness and comparison with the DAS beamformer.

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 unit weighting factor. The DAS equation also assumes that all signals travel along similar paths and also have 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 $\mathcal{A}$ and $\mathcal{A}'$, the more likely it is that the waves deviate from the straight-line path assumption, ergo the estimation of speed and subsequently the delay along that path will be wrong. S. A Shah Karam, et al. proposed a solution to this issue in their 2021 paper "Weighted delay-and-sum beamformer for breast cancer detection using microwave imaging" [9]. They suggested a weighting factor based on the transmitter-receiver distance (TRD_i) for the t^{th} observation:

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

α is a positive parameter that allows the above weighting factor to reward signals from nearby antennas while penalizing signals from antennas that are far away. This parameter is patient-specific and must be changed based on the homo- or heterogeneity of the breast tissue, with the paper using a value of $20cm$ for their tests. The paper also noted that since w_i is data-independent and focal point independent, a set of normalized weighting factors can be computed before hand and applied to the signals at collection time rather than processing time. 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, first proposed by Hooi Been Lim et al in 2008 [10], suggests 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. Upon comparison with the traditional DAS algorithm, the researchers found that the pair-wise multiplication before summation provided greater contrast and sharper results, greatly increasing the Signal-to-Clutter ratio of the image. Figure 8 shows the differences between the DAS and DMAS algorithms and most importantly, how the DMAS beamformer provides a greatly reduced noise floor.

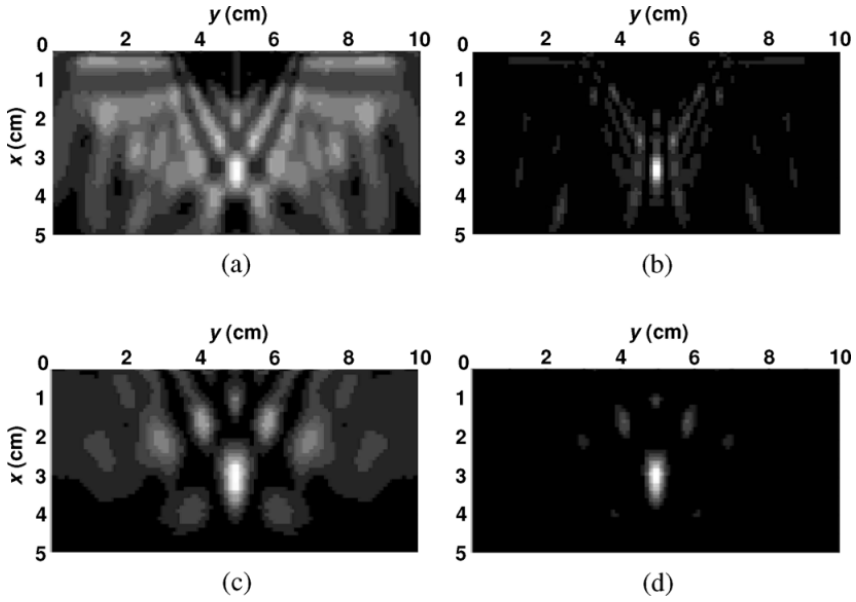


Figure 8: Comparason between DAS (a, c) and DMAS (b, d) Beamformers

The researchers never provided an explanation for the surprising effectiveness of the algorithm, leaving it 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 such a low noise floor. An explanation provided by Prof O'Loughlin posits that the pair-wise multiplication rewards signals that have a high degree of coherency while disproportionately penalizing incoherent signals. In essence, the coherent signals get amplified a lot more than incoherent signals [11]. G. Matrone et al. further reinforces this hypothesis by noting that the DMAS beamformer, after the signals are time aligned like in DAS, computes the autocorrelation of the receiver antenna with the auto terms removed [12].

Evaluating Programming Languages

A wide array of libraries aimed towards researchers and data scientists tend to be written in high-level languages such as Python and MATLAB. Their high-level syntax and open-source nature make them an attractive language to create and share libraries. This is backed up by statistics collected by JetBrains in 2022 which showed that out of the 23,000 people surveyed for their Python usage, 53% of these people used Python for some form of data science [**?**]. MATLAB also reports similar statistics, mentioning that their software suite is used by over 6500+ universities around the world for various purposes [**?**]. Julia is another programming language that has been gaining interest in research communities due to having high-level syntax like Python and MATLAB but also the performance that comes with low-level languages such as C. The Julia team conducted their own study of 1,329 individuals over a 3 month period in 2023 and found that 84% of people surveyed used Julia for research or teaching [**?**]. The prevalence of these programming languages in academia made them a potential candidate for MERIT.jl. The sections that follow will contain an analysis of the advantages and disadvantages of using each language.

2.1 Python

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 introduction of the PyPI Python Package Index in the late 2000's catapulted the new 20 year old language into the limelight by greatly simplifying the process of distributing libraries. Its success is evident with over 20.4TB worth of Python libraries being hosted by PyPI [**?**]. 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 [**?**]. For this reason, many libraries that require high performance in Python have a significant portion of their code written in C or C++, as demonstrated by their language breakdown in their respective GitHub repositories [**?**, **?**]. So developers who are concerned about speed have to be proficient in C and C++ as well as Python, creating a high barrier of entry which goes against MERIT.jl's goal of "easily extensibility".

Secondly, the Python Interpreter makes use of a runtime lock known as the Global Interpreter Lock (GIL) which makes parallel processing in Python difficult [**?**]. Python implemented the GIL as a consequence of their 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

are created, the count goes up, as references are deleted or reach the end of their scope, the count goes down. When this variable reaches zero, it becomes a flag for the object to be deleted. In some threaded applications, references can get created or deleted by multiple threads, causing this counter to be updated simultaneously. In some cases, this can cause the reference count to never reach zero leading to a memory leak, or reaching zero too early, prematurely deleting the object. To combat this, Python created the GIL to act as a lock on the interpreter. Any Python bytecode needs to first acquire a lock on the interpreter before it can be executed. This makes multithreading in Python difficult and slower than it would be in other languages. So for these reasons, Python was rejected as the software of choice.

2.2 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 analogs. MATLAB also has seen success in industry with one company estimating that it is used in over 57,811 companies [13]. However, MATLAB is an interpreted language like Python, meaning that it is slower than compiled code. 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 [14]. It should be noted that the same MATLAB code ran about 30.26x faster than their Python implementation, implying that even though both are interpreted languages, 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 against the open-source nature of MERIT.jl. The yearly licensing cost can be expensive for small research teams, barring them from contributing to the library. 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 absolute performance. For these reasons, MATLAB was also rejected as the software of choice.

2.3 Julia

The third and final language considered was the Julia programming language. Julia was developed by Jeff Bezanson, Stefan Karpinski, Viral B. Shah and Alan Edelman, and was first released to the public in 2012 [**?**]. The creators wanted a language that was as fast as C while also retaining the dynamism of high-level languages such as Ruby and Python. The Just In Time (JIT) compiler employed by the Julia runtime compiles the high-level language syntax into machine code allowing for large performance gains over other high-level languages. The aforementioned study found that Julia was only 1.47x slower than the comparable C++ implementation [14]. Added to this speed, Julia offers features that are not available in Python or MATLAB such as parametric polymorphism, multiple dispatch and efficient garbage collection. Julia also has native support for multiple parallel processing 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. These features allow Julia to have an "It Just Works!" ideology, where libraries written in other

supported languages as well as in Julia can all work together without significant additions of "glue-code". As stated before Julia has already garnered an interest in academia, this coupled with its performance made it a clear choice for MERIT.jl. The next chapter will discuss in detail the various Julia features that were used in the library and how they contributed to the overall goal of answering the research questions stated in the beginning.

Julia Specific Features

3.1 Multiple Dispatch

Multiple Dispatch is a programming paradigm in which a function can be 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 dispatched depends on a special argument placed before the function name. In almost all programming languages this is the variable name for that class. For example, consider the following code:

```
# Defined in one library
Class Dog:
    name::string
    function says(a::string):
        print("The dog, $self.name says $a")
    end
end

billy = Dog("Billy")

# Defined in another library
Class Cat:
    name::string
    function says(a::string):
        print("The cat, $self.name says $a")
    end
end

kate = Cat("Kate")
```

If one wanted to call the says from the Cat or Dog class in the single dispatch paradigm, one would have to write `billy.says("Hello World")` or `kate.says("Hello World")`. This concept fits well with object-oriented programming languages where classes are used to encapsulate concepts. However, a drawback of this system is that the compiler relies on the user to remember which methods belong to the class and which methods are callable. Also since both the Cat and Dog classes are defined in different libraries it is not clear how either developer could create a function where these classes interact with each other without having to create a separate third library that implements compatibility code.

Multiple and its particular implementation in Julia solves some of these issues. In Julia,

methods no longer belong to classes, instead like in C they simply belong to a particular library, also known as a "Module" in Julia. Functions defined in modules are usually exported such that they are available in the namespace of any other module that uses it. This allows modules to use and overload functions and structures from other modules in a way that is completely transparent to the end user.

```
# Defined in library A
struct Dog{
    name::string
}

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

# Defined in library B
struct Cat{
    name::string
}

function encounters(petA, petB)
    print("$petA.name encounters $petB.name and $meets(petA, petB)")
end

function meets(petA::Cat, petB::Dog)
    return "hisses"
end

function meets(petA::Dog, petB::Cat)
    return "barks"
end

# Overloading function from library A
function says(pet::Cat, a::string)
    print("The cat, $pet.name says $a")
end


# Someone else using both libraries
billy = Dog("Billy")
kate = Cat("Kate")

says(billy, "Hello World")
says(kate, "Hello World")

encounters(kate, billy)
encounters(billy, kate)
```

In the above code block, library B overloads the `says` function to accept a struct of type Cat. In addition to this, it also creates two new `meets` functions that handle the interaction between the Cat and the Dog types, as well as an `encounters` function. A third user can make use of both libraries as they did in the single dispatch case, but now they only have to call the function name, rather than the function name prepended with the class variable name. In the case of the `says` function call, the Julia compiler automatically dispatches the correct implementation based on the type of the input argument. This is further exemplified in the final two function calls. As stated before multiple dispatch is sensitive to both the type and order of the inputs. In the first call to the `encounters` function, the `meets` function on line 19 will be dispatched as the argument order was of type Cat and then Dog. Whereas in the second call to the `encoutners` function, the `meets` function defined on line 23 will be dispatched due to the reverse ordering of the types.

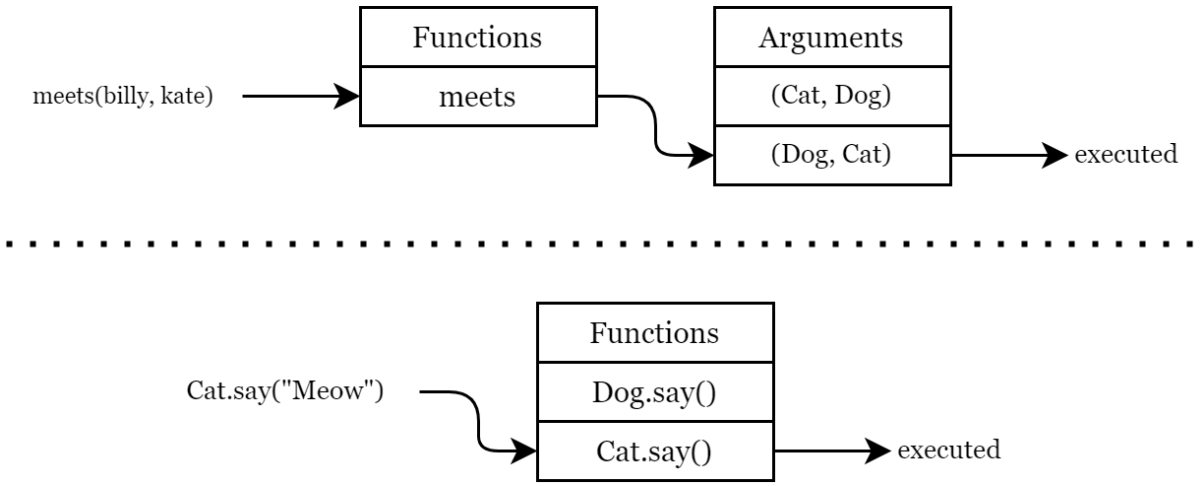


Figure 9: Multiple Dispatch (top) vs Single Dispatch (bottom)

3.2 Type Heirarchy

In Julia, all types 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 the tree, having parents but no children. Another notable difference between abstract types and concrete types is that abstract types cannot be instantiated, they serve only as nodes in the type graph. Shown in Figure 10, is the type hierarchy for the Integer type, abstract types are highlighted in red, whereas concrete types are highlighted in blue.

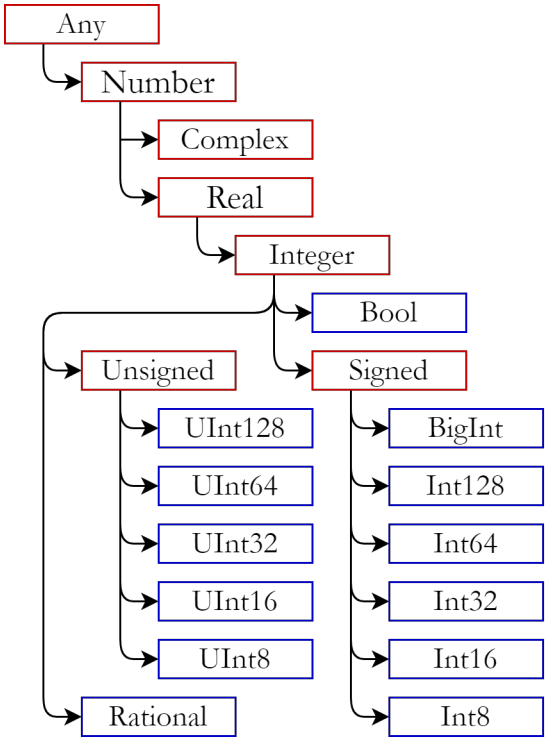


Figure 10: Type hierarchy for the Integer Type

The type hierarchy not only provides a way to logically organize types but also tightly integrates with the multiple dispatch system mentioned before. It is this pairing that allows Julia to fulfill its promise of easy and powerful extensibility. The following example helps illustrate this further.

```
# Defined in library A
abstract type Animal end

struct Cat <: Animal
    name::string
end

struct Dog <: Animal
    name::string
end

function encounters(petA::Animal, petB::Animal)
    verb = meets(petA, petB)
    println("$(petA.name) meets $(petB.name) and $(verb)")
end

meets(petA::Animal, petB::Animal) = "passes by."
meets(petA::Cat, petB::Dog) = "hisses"
meets(petA::Dog, petB::Cat) = "barks"

# Defined in library B
```

```
22 struct Rabbit <: Animal
23     name::String
24 end
25
26 whiskers = Cat("Whiskers")
27 chomper = Rabbit("Chomper")
28 encounters(whiskers, chomper)
```

Library A defines a type hierarchy with Cat and Dog being a subtype of the abstract Animal type. The library then implements a series of `meets` functions and an `ecounters` function, similar to the example in the previous section. However, unlike the example before, there is a new `meets` which accepts an argument of type Animal. Library B, importing library A, defines a Rabbit type that is a subtype of the Animal abstract type and then calls the encounters function from library A. The dispatch system cannot execute the `meets` function from lines 18, since `chompers` is not of type Dog. Instead, it executes the `meets` function on line 17, since both `whiskers` and `chompers` are valid Animal types. This highlights an important interplay between the type hierarchy and the multiple dispatch system, in that when dispatching, the Julia compiler will select the function that is most specific across all its input arguments. The developer of library A need not worry about all the other possible Animal types, or what their fields may contain, they can create a generic `meets` function that accepts any subtype of the Animal class and will execute successfully provided the subtyped class contains the fields being accessed. Neither developer has to worry about the completeness of the other's implementation, provided developer B follows the standards set by developer A, they can develop an extension to library A and trust that the two libraries will be comparable.

The concept of abstract types and subtypes is used heavily in MERIT.jl due to the constant pace of progression in the field. Currently, most of the research in microwave imaging is centered around breast imaging and breast cancer, however in 2013, a pilot study published in the International Journal of Biomedical Imaging found the use of microwave imaging to be beneficial when imaging transverse sections of the forearm [**?**]. As such the library needs to be flexible enough to adapt to novel imaging domains. MERIT.jl is centered around the Scan abstract type, from this, the BreastScan type is subtyped which holds all the information regarding the scan from a particular breast. In order to incorporate the aforementioned study, one would have to create a ForearmScan type, subtyped from Scan, which has a similar field name structure to the BreastScan type. In this way, MERIT.jl achieves extensive flexibility and expandability which would not be possible in other languages.

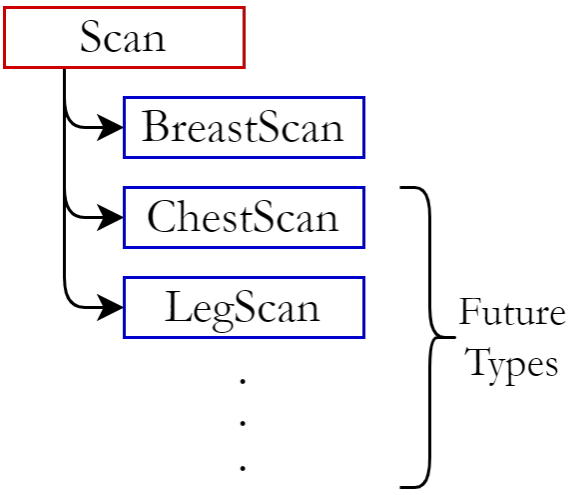


Figure 11: Current type hierarchy in MERIT.jl

3.3 Parametric Polymorphism

Parametric polymorphism refers to the programming paradigm where a function can be made generic to the input argument type. At compilation time, the compiler can strongly type the function based on the types of the input arguments. Consider the simple example of an `addtwo(x)` function.

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

Instead of specifying a concrete type for `x`, it is parametrized by the variable `T` which has an abstract type restriction on it. This implies that `T`, and by extension `x`, can by any type that is a subtype of the Real abstract type. If parametric polymorphism was not a Julia feature, this function would have to be duplicated 16 times to account for the various concrete types that are a subtype of Real (see Figure 10).

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 will be loaded and can therefore set the internal data type of the struct. It would be impossible to support all 4,176 different permutations of every type pairing, however, with parametric polymorphism, each 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, creating an intuitive and easy coding experience.

3.4 Type Stability

Type Stability is a coding discipline that Julia recommends all code use. A function whose contained variables have a consistent type for its lifetime is considered to be "type stable" or to have "type stability". Another way to think about this, is that if the output type

depends only 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
```

Both functions have a variable, `x` which is divided 10 times by random floats and is returned at the end of the function. However, based on the description above, the first function is considered to be "Type Unstable" whereas the second function is considered to be "Type Stable". This is because in the first function the type of `x` changes in the lifetime of the function (turns from Int64 to Float64). However in the second function, the data type of `x` stays as a Float64 throughout the lifetime of the function. The notion of type stability is important as it allows the Julia compiler to perform optimizations based on the known data type of the variables at compile time. For type unstable functions, their unstable variables default to type Any, ergo forgoing any optimizations. These optimizations are evident when benchmarking the functions above. This was performed using the `@benchmark` macro from the BenchmarkTools module. Both functions were run multiple times to ensure that only the compiled versions of the functions were executed rather than the interpreted version. These benchmarks showed that on average the type unstable function ran 1.87x slower than the type stable function (573.355ns and 306.178ns respectively). These performance gains become significant when considering the many millions of function calls required to generate an output in MERIT.jl. Consider just the distance calculations required between an $8cm$ radius breast whose domain is discretized with a resolution of $0.25cm$ and an antenna array made of 60 antennas; this equates to roughly 4.2M calculations alone for each image. Performing this on a type unstable function would take far too long to compute, rendering the library unusable. Herein lies one of the drawbacks of writing Julia code, namely the ease at which an inexperienced developer can write type unstable functions. In the simple case above, the cause for the type instability is evident, however, with larger more complex functions it can be hard to narrow down the cause for the instability. Creating performant code in these situations requires advanced knowledge of the language, raising the barrier to entry. However, the language offers the `@code_warntype` macro that can analyze sections of code and indicate, but not fix, places of instability where the compiler fails to infer the data type of the variable. This tool can considerably benefit developers of MERIT.jl to ensure that any extensions written for the language remain in line with its philosophy of performance.

3.5 Closure

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

```
function addX(x)
    scalar = x
    function calc_(a)
        return a + scalar
    end
end

add5 = addX(5)
add7 = addX(7)

add5(2)  # Will return 7
add7(10) # Will return 17
```

Defined above is a function called `addX` which accepts an input `x` and assigns it to a variable `scalar`, before constructing and returning the function `calc_`. This can be beneficial as it allows developers to create a family of functions that are parametrized by their input variable. MERIT.jl makes use of closure in the `get_delys` in the Breamform.jl file (for more information see the GitHub link in the Introduction). Earlier sections showed how ϵ is a free parameter in the beamforming equations, one that researchers would be changing frequently to find the optimal parameter for each particular scan. Implementing this feature fulfills one of the other tenets of MERIT.jl which was to create flexibility and customizability in the functions where it was required. It must be noted that the captured variable, in the above case `scalar`, must never be reassigned, otherwise, the compiler will not be able to infer the data types leading to type unstable code, and by extension all the negative consequences mentioned in the previous section.

3.6 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, it is simple to add two numeric types, but performing the same operation on two strings would yield an error, since the summation operator is not defined for two string types. This offers some level of protection against illegal operations, but there are many cases where the type of the variables agree with the operation being performed, but their semantic meaning disagrees. The example below will illustrate this idea.

```
function calcSpeed(dist::Vector{T}, t::Vector{T}) where {T <:
    Real}
    return dist ./ t
end

#####################################################
distance = rand(1, 100)
```

```
time = rand(1, 100)
calcSpeed(distance, time)    # Will compute the speed

#####################################################
distance = rand(1, 100)
time = rand(ComplexF64,1, 100)

# Will throw an error since there is a type mismatch
calcSpeed(distance, time)

#####################################################
distance = rand(1, 100)
time = rand(1, 100)

# Will compute the wrong answer, but no error gets thrown
calcSpeed(time, distance)
```

In the code block above `calcSpeed` accepts a vector of distance and time and returns the associated speeds. Executing line 8, as expected returns the calculated speeds correctly. Executing line 12 will throw a type error since ComplexF64 is not a subtype of Real. Line 22 however, will execute and return successfully even though the wrong answer is returned, as the arguments were provided in the wrong positional order. This is because compilers can only make inferences regarding the legality of a statement based solely on the concrete types of its arguments rather than their semantic meaning in the context of the function being executed. However, the chance of variables being incorrectly passed positionally can be negated 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 act as wrappers around a numerical concrete type. This way when line 22 is executed, instead of returning an incorrect answer, the compiler will raise an error as the "time" and "distance" types were provided in the wrong order.

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 numerical types respectively and serve the purpose of being a 3D and 2D point.

```
abstract type Point end

# xyz can be any data type that is a subset of Real
mutable struct Point3{T <: Real} <: Point
    x::T
    y::T
    z::T
end

mutable struct Point2{T <:  Real} <: Point
    x::T
    y::T
end
```

Due to the custom nature of these data types, the inbuilt operators could not be used on them. So in addition, MERIT.jl had to extend the in-built operators using the concepts of multiple dispatch and parametric polymorphism as mentioned before such that these types could be useful. The full suite of implemented operators can be found in the GitHub repository. Every function in the library that needs to work with points accepts a collection of a Points subtype rather than a collection of numbers. This way no other collection of numbers can be erroneously passed in place of the Points subtypes. Some edge cases still exist however, there is nothing stopping a user from incorrectly passing a collection of points describing antenna locations to an argument which is for points from the imaging domain. Even though this issue still exists, clear and easy to understand documentation should make this a nonissue. Due to time constraints on the project, the entire library could not be made "strongly" typed, however, the Points type acts as s template that could be used to eventually strongly type the entire library

3.7 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 reference 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 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

    # Create a delay function
    function calc_(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 counterpart. 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 in previous sections, 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

4.1 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 12.

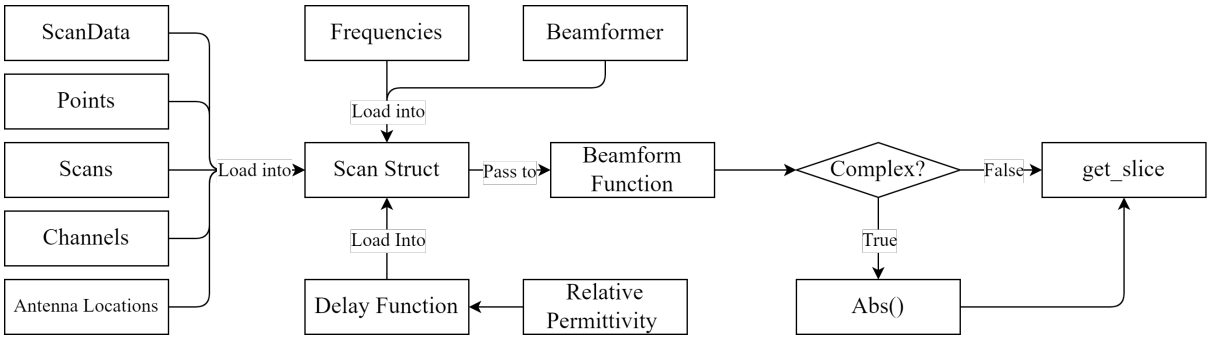
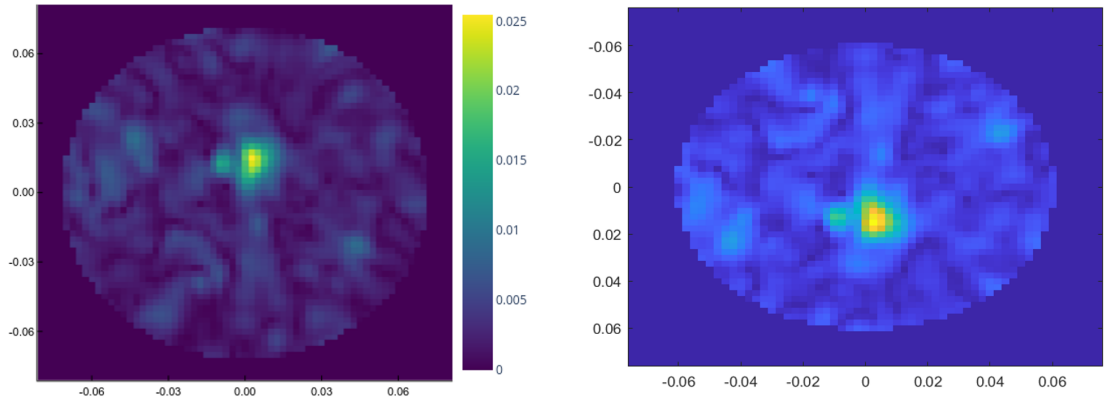


Figure 12: The entire MERIT.jl Workflow

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 accessible library.

4.2 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 13. 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

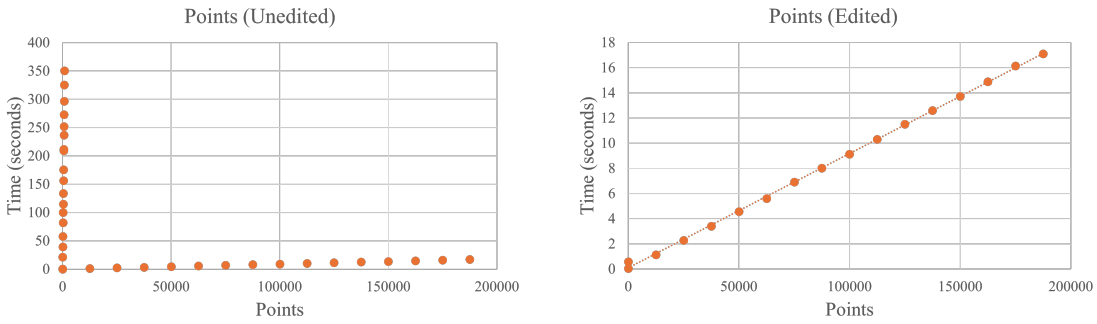
(b) MATLAB Output

Figure 13: A comparison between Julia and MATLAB output

4.3 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 [15]. 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.

Shown above in Figures 14, 15, 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 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) Unedited Points (b) Edited Points

Figure 14: Runtime for increasing Points

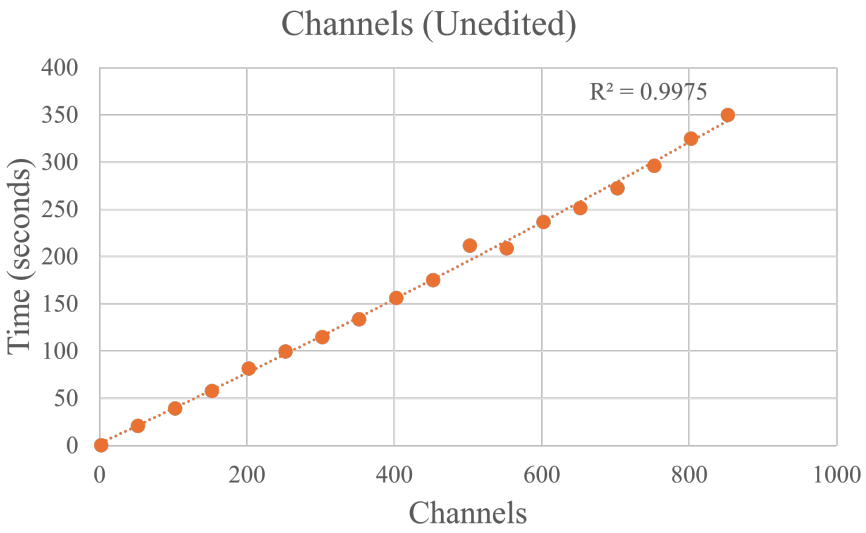
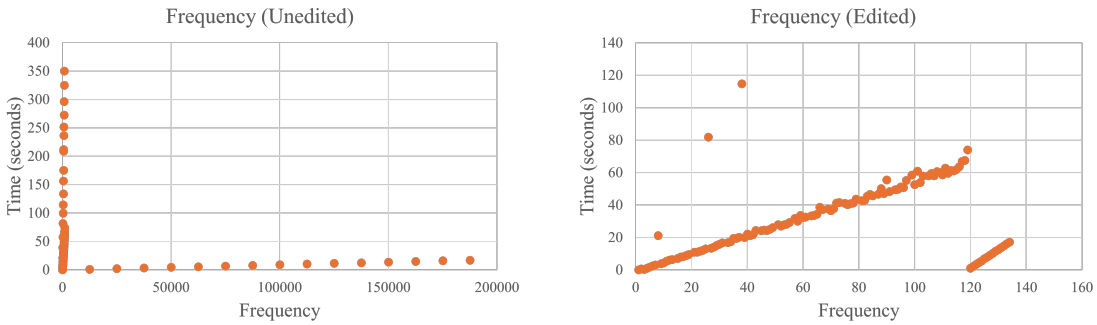


Figure 15: Runtime for increasing Channels



(a) Unedited Frequencies (b) Edited Frequencies

Figure 16: Runtime for increasing Frequency divisions

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 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 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

5.1 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.

5.2 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:

$$I_{\epsilon_i}(r) = \mathcal{G}(\mathcal{S}) \left[\sum_{\Omega} \sum_{\mathcal{A}'} \sum_{\mathcal{A}} S_{a,a'}[\omega] e^{j\omega \tau_{\epsilon_i,a,a'}(r)} \right]^2 \tag{5.6}$$

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.

5.3 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 that 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 [16]. 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.

Conclusions

Microwave imaging is making waves in the medical field, with the potential to create cheaper and safer imaging modalities. With many systems at various stages of clinical trials, it is important now more than ever to create a library that can encourage compatibility and interoperability. MERIT.jl achieves this through the use of the Julia programming language. In its current state, it can ingest data from most systems and perform frequency domain beamforming to produce an image that is near identical to its MATLAB counterpart. By making use of Parametric Polymorphism, Multiple dispatch, Type Safety, Type Stability and Closure, MERIT.jl has demonstrated powerful extensibility and flexibility while also remaining performant. MERIT.jl's open-source nature, near-limitless customizability and high-level syntax make it an appealing choice to both beginner and advanced researchers alike.

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] D. O'Loughlin, M. A. Elahi, E. Porter, A. Shahzad, B. L. Oliveira, M. Glavin, E. Jones, and M. O'Halloran, "Open-source software for microwave radar-based image reconstruction," in 12th European Conference on Antennas and Propagation (EuCAP 2018), pp. 1–4.

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

[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.

[8] D. O'Loughlin, B. L. Oliveira, M. A. Elahi, M. Glavin, E. Jones, M. Popović, and M. O'Halloran, "Parameter Search Algorithms for Microwave Radar-Based Breast Imaging: Focal Quality Metrics as Fitness Functions," vol. 17, no. 12.

[9] S. A. Shah Karam, D. O'Loughlin, B. L. Oliveira, M. O'Halloran, and B. M. Asl, "Weighted delay-and-sum beamformer for breast cancer detection using microwave imaging," vol. 177, p. 109283. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0263224121002864

[10] H. Been Lim, N. Thi Tuyet Nhung, E. -P. Li, and N. Duc Thang, "Confocal Microwave Imaging for Breast Cancer Detection: Delay-Multiply-and-Sum Image Reconstruction Algorithm," vol. 55, no. 6, pp. 1697–1704.

[11] D. O'Loughlin, B. L. Oliveira, M. Glavin, E. Jones, and M. O'Halloran, "Comparing Radar-Based Breast Imaging Algorithm Performance with Realistic Patient-Specific Permittivity Estimation," vol. 5, no. 11.

[12] G. Matrone, A. S. Savoia, G. Caliano, and G. Magenes, "The Delay Multiply and Sum Beamforming Algorithm in Ultrasound B-Mode Medical Imaging," vol. 34, no. 4, pp. 940–949.

[13] Companies using MATLAB . Enlyft. [Online]. Available: https://enlyft.com/tech/products/matlab

[14] S. B. Aruoba and J. Fernandez-Villaverde, "A Comparison of Programming Languages in Economics: An Update," p. 4. [Online]. Available: https://www.sas.upenn.edu/~jesusfv/Update_March_23_2018.pdf

[15] j. . BenchmarkTools.jl. BenchmarkTools. [Online]. Available: https://juliaci.github.io/BenchmarkTools.jl/stable/

[16] J. . JuliaGPU. [Online]. Available: https://juliagpu.org/