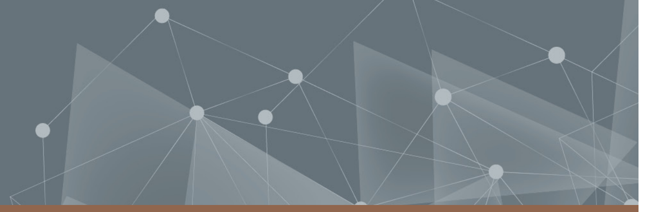
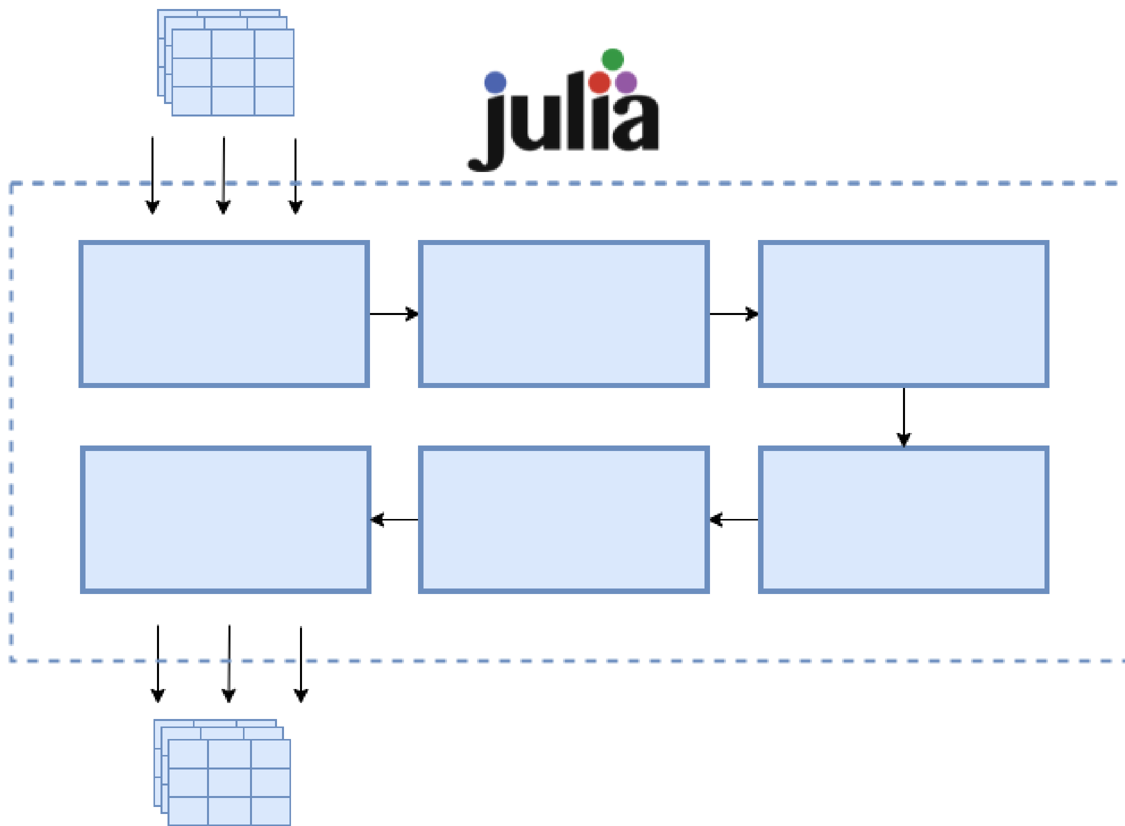
CHALMERS
UNIVERSITY OF TECHNOLOGY





The Julia programming language for embedded high-performance signal processing in radar systems

Master's thesis in System, Control and Mechatronics

ANTON KARLSSON

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021

www.chalmers.se

Master's thesis 2021

ANTON KARLSSON

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2021

ANTON KARLSSON

© Anton Karlsson, 2021.

Supervisor: Per Ekström, Erik Norlander and Anders Åhlander, SAAB
Examiner: Tomas McKelvey, Department of Electrical Engineeing

Master's Thesis 2021
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

Julia master thesis
The Julia programming language for embedded high-performance signal processing in radar systems at SAAB
Anton Karlsson
DEPARTMENT OF ELECTRICAL ENGINEERING
Chalmers University of Technology

Abstract

Julia is an emerging open-source programming language developed at the Massachusetts Institute of Technology (MIT). The language promises high-level expressive syntax and speed comparable to that of C/C++. This is achieved via the use of a JIT (Just In Time) compiler. In this thesis, Julia is examined for high-speed signal processing at SAAB. The case considered in this thesis is an AESA signal processing chain. Throughout this thesis, Julia is mainly compared with Matlab which is the language that is typically used for prototyping today. It is also compared with C/C++ which are typically used in the end product for performance and reliability. It is further examined how functionality developed in Julia can be embedded into a C framework. Julia shows great potential for the future, yet it still has some room to mature.

Keywords: Julia, Matlab, C, Radar, Signal Processing, Embedding.

Sammanfattning

Julia är ett snabbt växande och fritt tillgängligt programmeringsspråk som har utvecklats på Massachusetts Institute of Technology (MIT). Språket utlovar en uttrycksfull hög-nivå syntax med prestanda jämförbar med C/C++. Den här prestandan kommer från Julias *Just In Time* (JIT) kompilator. I den här uppsatsen har programmeringsspråket Julia undersökts för användning inom högpresterande signal-behandling på SAAB. Fallet som testats är en signal-behandlings kedja för en enkel AESA radar. Genom det här projektet har Julia främst jämförts med Matlab vilket frekvent används vid prototyp utveckling på SAAB idag. Julia jämförs också med C/C++ som till stor grad används vid implementation av slut produkt för prestanda och tillförlitlighet. Det har även undersökts hur funktionalitet utvecklad i Julia kan byggas in i andra språk. Julia påvisar stor potential för framtiden men det finns visst utrymme för språket att mogna.

Acknowledgements

I would like to acknowledge my supervisors at SAAB Anders Åhlander, Erik Norlander and Per Ekström that has guided me through this project and provided me with support. I would further like to acknowledge significant assistance with the C++ example from Per Ekström. I would also like to acknowledge my supervisor at Chalmers Tomas McKelvey for valuable input and support.

Anton Karlsson, Gothenburg, June 2021

Contents

1 Introduction **1**

1.1 Background . . . 1

1.1.1 Signal processing development process . . . 1

1.1.2 Julia the programming language . . . 1

1.2 Purpose . . . 2

1.3 Objectives . . . 2

1.4 Scope . . . 2

2 Julia features **3**

2.1 Julia types . . . 3

2.2 Multiple Dispatch . . . 3

2.3 Just in time compiler . . . 4

2.4 Mutation . . . 4

2.5 Garbage collection . . . 5

2.6 Benchmarking . . . 6

2.7 Signal processing packages . . . 6

2.8 Threading . . . 6

2.8.1 Static threading . . . 7

2.8.2 Dynamic threading . . . 7

2.8.3 Known issues with threading . . . 7

2.9 Embedded Julia in C . . . 8

2.9.1 Loading and calling functions . . . 9

2.9.2 Rooting variables . . . 9

2.9.3 Shared Memory . . . 10

3 Radar Signal Processing **13**

3.1 Pulse-Doppler Radar . . . 13

3.2 Interference . . . 14

3.2.1 Clutter . . . 14

3.2.2 Jamming . . . 14

3.2.3 Noise . . . 14

3.3 AESA-Radar . . . 14

3.4 Data acquisition . . . 15

3.5 Digital beamforming . . . 16

3.6 Pulse Compression . . . 18

3.7 MTI . . . 19

3.8 Doppler Processing 21
3.9 Equalization 22
3.10 Detection 22
3.11 STAP 22

4 Results and Discussion **27**
4.1 Beamforming 27
4.1.1 Discussion 28
4.2 MTI 28
4.2.1 Discussion 28
4.3 Embedding 29
4.3.1 Serial test 29
4.3.2 Threaded test 29
4.3.3 Discussion 30
4.4 SLC test 30
4.4.1 Runtimes with AMD 30
4.4.2 Runtimes with INTEL 31
4.4.3 Discussion 31
4.5 Garbage Collection 31
4.5.1 Overhead of calls to GC 31
4.5.2 Managed GC vs unmanaged GC 32
4.5.3 Discussion 33
4.6 Compared to Matlab 34
4.7 Compared C/C++ 34
4.8 Two language problem 35

5 Conclusion **37**

6 Future Work **39**

A Appendix **43**
A.1 Package environments 43
A.2 System image 44
A.3 Code 45
A.3.1 AESA Julia 45
A.3.2 Embedded AESA 52

Glossary

AESA	*Active Electronically Scanned Array*
BLAS	*Basic Linear Algebra Subprograms*
CFAR	*Constant False Alarm Rate*
DBF	*Digital Beam Forming*
DFB	*Doppler Filter Bank*
GC	*Garbage Collection*
Julia	Programming language developed at MIT
LLVM	*Low Level Virtual Machine*
MKL	*Math Kernel Library* Linear algebra vendor provided by Intel
OpenBLAS	Open source linear algebra vendor provided as default in Julia
PC	*Pulse Compression*
PRF	*Pulse Repetition Frequency*
PRI	*Pulse Repetition Interval* Time interval of a sampled pulse
RADAR	*Radio Detection And Ranging*
REPL	*Read Eval Print Loop* is an interactive command prompt for Juila.
SAAB	Swedish company active in various branches of the defence industry
SLC	*Side Lobe Cancellation*
STAP	*Space Time Adaptive Processing*

1
Introduction

Radar systems are growing ever more sophisticated with high-tech hardware and smart software solutions making today's systems more versatile and effective than ever. These advancements however come with increased complexity and processing requirements as the number of computations has increased exponentially with products like Active Electronically Scanned Array (AESA) radar systems. As is indicated by its name the AESA radar consists of an array of radar elements that all transmit and receives electromagnetic signals, by exploiting constructive interference it can synchronize signals into a coherent beam that can be aimed electronically without any mechanical movement. SAAB is a company that is at the forefront of radar technology. In this thesis the programming language, Julia[1] will be explored and assessed for possible use in radar development at SAAB.

1.1 Background

1.1.1 Signal processing development process

Signal processing embedded in hardware is traditionally done in compiled languages such as C/C++ which run very efficiently but they are not well suited for interactive development of new models and concepts as they require to be compiled and has stiffer constraints. Model development is today typically done in expressive and interactive high level languages like Matlab and Python where it is convenient to test features, plot results etc. When a model is good enough for testing on the hardware it would have to be translated into a compiled language like C/C++ through substantial labour which means the work is done twice and the cycle would fall victim to the two language problem. Not only is this labour intensive for initial development but also for continuous maintenance of the two models that require continuity with each-other to remain relevant. The custom of maintaining models in two languages is however not only bad as it is valuable for verification purposes.

1.1.2 Julia the programming language

The programming language Julia[1] was developed at MIT and launched to the public in 2012. Julia is a dynamically typed high-level language with computation efficiency comparable to that of C/C++. This efficiency is achieved with a just in time (JIT) compiler which compiles functions into machine-code after first execution and then keeps them in memory which makes subsequent executions as quick as in any compiled language. In a study performed at NASA [16] benchmarks for basic

operations in several languages was compared. In these tests, Julia was shown to be very competitive even compared with C. Julia performed well for all benchmarks and was notably not bad at any of the tests highlighting the general-purpose suitability of Julia. One key feature of Julia is that the syntax closely resembles mathematical formulas such that the step from a concept to high performing implementation isn't too large. Julia is also garbage collected which means that the developer does not actively have to allocate and free memory. These properties have made Julia a popular language in academic circles during the last years as it has been possible to streamline work.

1.2 Purpose

The purpose of this project is to investigate how well suited Julia is for signal processing in radar systems. The hope is that Julia for some applications can work both as a language for high-level modelling where Matlab is typically used today as well as high-performance implementation where C/C++ is typically used.

1.3 Objectives

The objective is to investigate how useful Julia is for the development of radar systems. Usefulness is in this case determined by the performance of the end product but also in terms of decreased complexity in the development process. The following questions will be attempted to be answered.

- How can an AESA signal processing chain be implemented in Julia with good performance on available hardware and does it work in real-time?
- Is Julia viable for embedded systems and how can that be realized?
- What are the pros and cons of Julia relative to Matlab in terms of development convenience and execution performance?
- How do the features of Julia compare to C/C++, how can that better the development process of embedded signal processing chains?

1.4 Scope

The thesis is to be performed during a 20 week period in the spring of 2021 and is performed by one person. The project will be limited to simulations and no tests with real radar systems will be conducted. This thesis is not concerned with radar design and will thus not necessarily have parameters that would make up an effective radar. The parameters used in this thesis are not based on any existing system on SAAB as such information is classified. The complete model will only be implemented in Julia, reference benchmarks in Matlab and C/C++ will only be done for some choice functions. Security aspects of Julia such as protection of source code will not be covered in this thesis.

2
Julia features

In this section some Julia features related to the thesis will be presented, for further reading check out the Julia manual at https://docs.julialang.org/en/v1/.

2.1 Julia types

Julia employs an abstract type system where all types derive from the super-type *any* [31]. In Figure 2.1 it is illustrated how some types branch out from type *any* down to specific types that can actually be represented in memory that are called bit-types. This way of representing types is useful as it enables developers to write functions that works for more than a single configuration of bit-types.

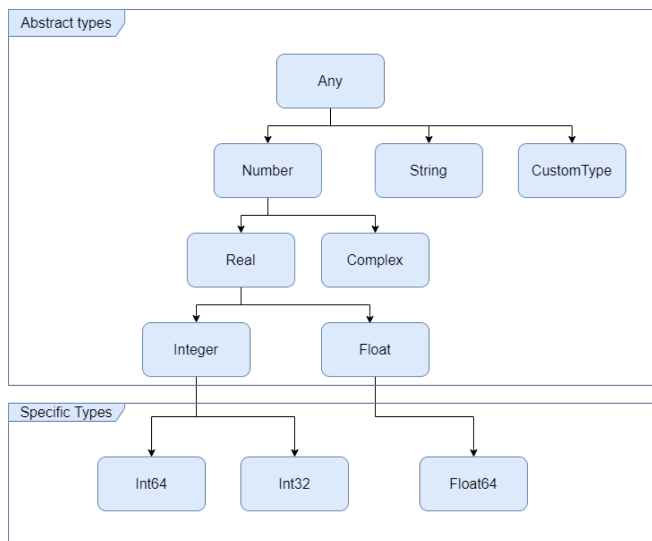


Figure 2.1: Example of type hierarchy

2.2 Multiple Dispatch

Julia uses multiple dispatch [1] which means that functions in Julia can be defined for multiple input configurations where types and number of arguments are used to decide which version of the function should be called. During the runtime the function is built for every encountered input configuration, a function with a specified input configuration is in Julia referred to as a method. The difference between

multiple dispatch and single dispatch is that the method is determined by all arguments provided to a function not just one argument as is common in object-oriented languages.

```
#single dispatch method is determined by the type of x1 only
x1.function_name(x2,x3,x4)

#multiple dispatch method is determined by all input arguments
function_name(x1,x2,x3,x4)
```

2.3 Just in time compiler

Julia uses a just in time (JIT) compiler which compiles functions just ahead of the first execution and is for this reason commonly referred to as a just ahead of time (JAOT) compiler. The Julia compiler converts high-level syntax into a lower intermediate representation (IR) targeting LLVM [30]. LLVM can from this IR produce optimized native code for a targeted platform. The compiled native code is then kept in memory which makes subsequent executions as fast as in any compiled statically typed language. With this compiler setup the Julia team is able to have a narrow scope towards the LLVM IR and let LLVM handle much of the heavy lifting to reach as many platforms as possible.The downside of JIT compilation is that a significant amount of time is required to compile functions to native code during runtime that results in long startup times. However, a snapshot of a Julia session can be saved in a system image that contains the compiled functions and be used to speed up startup times see Appendix A.2. JIT compilers are to mention a few used by Matlab and Python with the package Numba [22]. What sets Julia apart is that the language was developed with JIT compilation in mind whereas the previously mentioned languages adapted it to their framework.

2.4 Mutation

Julia functions can be divided into mutating and non-mutating ones where the mutating functions are denoted with an exclamation mark at the end. In Source Code 2.1 is an example of how this can look for a simple function.

Source Code 2.1: Mutation example

```
function mult(A:: Array{<:Number,2},B::Array{<:Number,2})
    # Returns the results as a new variable
    return A*B
end
function mult!( A::Array{<:Number,2},
                B::Array{<:Number,2},
                C::Array{<:Number,2})
    # Mutates the preallocated variable C
```

```
    C .= A*B
end
```

Mutation is important for numerical computations as one can reuse memory and avoid excessive allocations that leads to additional work for the garbage collection. Mutation of this sort is not possible with Matlab functions and if one translates code from Matlab into Julia one should be wary not to overwrite memory without intending to.

2.5 Garbage collection

Dynamic data structures such as the conventional array in Julia are allocated memory with some reference variable pointing to it. If such a variable is unreferenced the memory that was held by the said variable is no longer reachable and considered to be garbage. As a computer has a finite amount of memory available the garbage has to be freed somehow. In some languages like C/C++ this is done manually by first explicitly allocating memory and then freeing it when it is no longer being used. In Julia however, the garbage is removed with an automatic garbage collection that sweeps the memory when some condition for this is met. How a garbage collector works differ from language to language. In Julia its key characteristics [13][17] can be summarized as :

- *Mark and sweep tracing:* When the garbage collector runs it starts at a root and traces an object graph where the reachable objects are marked, all objects that are not marked will be deemed unreachable and are "swept away" i.e the occupied memory is freed.
- *Non-compacting and non-moving:* Julias garbage collector will not compress, copy or move any memory.
- *Generational:* Typically recently allocated objects become garbage faster than old ones according to the "generational hypothesis"[19]. Because of this Julia implements frequent sweeps that just examines young objects and seldom perform full sweeps as they are costly and often unnecessary.
- *Stop the world:* When garbage collection is active everything else stops including work on multi-threaded processes.

```
# Sweep young objects
GC.gc(false) # full=false

# Full sweep
GC.gc() # full = true (default)

# Turn of GC
GC.enable(false)

# Turn on GC
GC.enable(true)
```

2.6 Benchmarking

Julia offers several useful tools for benchmarks some light ones defined in Julias standard library. For the purpose of robust measurements, the package Benchmark-Tools[11] is a good option. With this package, functions can be tested with multiple iterations to get robust results. The theory behind the package is covered in [3] and contains an analytical approach to chose the number of iterations to get statistically accurate benchmarks.

Table 2.1: Functions and macros for timing and benchmarks

time() :	Returns time in seconds since epoch
time_ns() :	Returns time in nano seconds undefined time 0 and can thus only used to measure a high accuracy difference in time.
@time :	Prints elapsed time, allocations, memory usage, percentage of time spent on garbage and as of v1.6.0 percentage of time that is used for precompilation [14].
@elapsed :	Returns time to execute a function.
@btime :	Light benchmark with several iterations, similar print to @time but more accurate
@benchmark :	Heavier benchmark with more iterations prints thorough benchmarking statistics

2.7 Signal processing packages

Julia offers some signal processing packages most noteworthy:

- FFTW[12] that contains efficient fft functions implemented in C this package requires a license fee for commercial use.
- DSP[4] contains useful window functions, filter functions and various signal processing utility functions.

2.8 Threading

Threading [21] is a convenient parallelization method available in Julia where multiple threads can work on shared memory where no copying of data between processes is necessary. The downside is that shared memory might lead to a data race[26] if one attempts to access the same memory from different threads concurrently. As most radar signal processing operations has a single instruction multiple data (SIMD) character it is possible to structure algorithms such that it is possible to parallelize without causing a data race. There are two main approaches to threading "Static threading" and "Dynamic threading" [27].

2.8.1 Static threading

Static threading is rather intuitive where the "@threads" macro is put in front of a for loop that splits the loop iterations between different threads. Each thread can then perform some computation as long as the operations are not sequentially dependent. It is important to avoid writing data to the same memory from multiple threads as that would cause a data race [26]. If one has to write to the same memory it is possible to temporarily lock a piece of memory to one thread however this comes with significant overhead and is not used within this thesis. Another limitation of static threading is that threads are assigned their workload before any computations start meaning that if Julia has enabled 4 threads and 3 threads finish their computations quickly they will be idle waiting for the last thread to finish. This issue would be especially problematic in cases where processes have convergence criteria which might lead to large differences in computation time.

```
@threads for i in 1:length(data_in)
    data_post_foo[i] = foo(data_in[i])
    data_post_bar[i] = bar(data_post_foo[i])
end
```

2.8.2 Dynamic threading

Dynamic threading is based on spawning tasks that will be picked up and carried out asynchronously by whichever thread is available. When the spawned task is completed one can access the return value. The same computation as above can be carried out dynamically by spawning tasks with the following.

```
tasks = [Threads.@spawn bar(foo(data_in[i])) for i in 1:length(data_in)]
#possible to do other computations asynchronously here
data_post_bar = [fetch(t) for t in tasks]
```

2.8.3 Known issues with threading

An open issue with threading in Julia is that it is not compatible with the internal threading in linear algebra operations provided by basic linear algebra suppliers (BLAS). Thus it is advisable to set the BLAS threads to 1 to avoid significant drops in performance. This can be done directly from Julia with the following.

```
using LinearAlgebra
BLAS.set_num_threads(1)
```

Threading of fft operations are however compatible even though they also have internal threading [23].

2.9 Embedded Julia in C

In the following section basics in how to embed Julia in C is presented [6]. There is an API to access and run Julia from C where a Julia session is initiated from within a C program. It is from there possible to interact with Julia just like one would from the Julia REPL using *jl_eval_string("<Julia command>")* except it would be called from within a statically compiled C program. The call to Julia will return a pointer to some Julia value, if the type is known it is possible to extract the output data from within C. The returned Julia value is however not safe as that piece of memory might be freed by the GC as it is not aware if it is accessed from outside Julia. In the example in Source Code 2.2 the accessed value is exempted from GC by calling JL_GC_PUSH1(&val). Measures that can be taken to make values safe from GC is covered in detail in Section 2.9.2. In the example it is further illustrated how to perform some basic operations like how to load package manager and activate a project environment which are explained in Appendix A.1.

Source Code 2.2: Basic Embedding

```
#include <julia.h>
#include <stdio.h>
JULIA_DEFINE_FAST_TLS() /* only define this once,
in an executable (not in a shared library) if you want fast code.*/
int main(int argc, char *argv[])
{
    /* required: setup the Julia context */
    jl_init();

    /* Write to your Julia session from C */
    jl_eval_string("<some Julia command>");

    /* Start a project environment */
    jl_eval_string("import(Pkg)");
    jl_eval_string("Pkg.activate(\"WorkingProject\")");

    /* Run code from a Julia file */
    jl_eval_string("include(\"main.jl\")");

    /* Accesing a return value from the Julia as a pointer
    that one can unbox if the type is known*/
    jl_value_t* ret = jl_eval_string("sqrt(2.0)");
    JL_GC_PUSH1(&ret); // Exempt from GC

    if (jl_typeis(ret, jl_float64_type)) {
        double ret_unboxed = jl_unbox_float64(ret);
        printf("sqrt(2.0) in C: \%e \n", ret_unboxed);
    }
    else {
```

```
        printf("ERROR: unexpected return type from sqrt(::Float64)\n");
    }
    JL_GC_POP(); //Remove GC exemption


    /* Exiting the Julia session */
    jl_atexit_hook(0);
    return 0;
}
```

2.9.1 Loading and calling functions

With the *jl_eval_string* statement it is possible to run embedded Julia code, it is however of limited use as it does not allow for input arguments. For this reason it is possible to access Julia function-handles from the embedded environment and calling functions with arguments using *jl_call*. When loading a Julia function from C it is necessary to know in which module it is contained of which there are three standard one's core, base and main [29]. The core module holds all the built-in functionality of Julia, the base module holds basic functionality and the main module holds all functions loaded from third-party packages and functions defined by the user. In Source Code 2.3 it can be seen how one can run a Julia script that defines a function which is then accessed from the main module and called with arguments. A more in depth example of this can be seen in Appendix A.3.2.

Source Code 2.3: Load functions

```
/* Loads function defined in base  */
jl_function_t *sqrtfunc = jl_get_function(jl_base_module, "sqrt");

/* Run script containing function myfunc */
jl_eval_string("include(\"myfunc.jl\")");

/* Loads self defined functions in main  */
jl_function_t *myfunc = jl_get_function(jl_main_module, "myfunc");

/* Call self defined function  */
jl_value_t* val = jl_call(myfunc, args, nargs)
```

2.9.2 Rooting variables

As previously mentioned Julia variables are accessed as pointers from C which creates an issue as Julia's garbage collection will not be aware that a Julia variable is referenced in C. The GC might for this reason at any time sweep in and free said memory, any attempt to access this memory will then crash the program. To keep memory safe all Julia values within C needs to be rooted in Julia. This can be done by explicitly telling GC to temporarily exempt a set of references. The

exemption can however only be done for one set of references within each scope. If one wants to keep variables safe for longer the suggested semantics are to store the references within a Julia dictionary as illustrated in Source Code 2.4. Values that are stored in this dictionary must be mutable objects, thus if they are immutable one is required to wrap them in a mutable container like *RefValue{Any}* with the use of *jl_new_struct* as is illustrated in the example. If objects are already mutable one can directly store them in the dictionary without wraping them.

Source Code 2.4: Rooting

```
// Hash to store pointers in global scope to keep data safe from GC
jl_value_t* refs = jl_eval_string("refs = IdDict()");
jl_function_t* setindex = jl_get_function(jl_base_module, "setindex!");
jl_datatype_t* reft = (jl_datatype_t*)jl_eval_string("Base.RefValue{Any}");


jl_value_t* val = jl_eval_string("<some Julia command>");
{
JL_GC_PUSH1(&val);
jl_value_t* rval = jl_new_struct(reft, val);
jl_call3(setindex, refs, rval, rval);
JL_GC_POP();
}
```

2.9.3 Shared Memory

Julia and C can operate on the same memory, making embedded use of Julia feasible. Figure 2.2 shows how memory is instantiated in C and then accessed as an array in Julia by pointing at the memory and providing the specific type and dimensions of the array. Note that creating the Julia array requires knowledge about the Julia types as well as a tuple of dimensions. The most basic specific Julia types such as Float64 are predefined within *julia.h* as jl_float64_type. The number of predefined types is for obvious reasons finite and do not include for example complex types which are instrumental in signal processing. Such types can however be accessed as the return pointer of a *jl_eval_string* statement. The same goes for any desired Julia object one wants to access. In Source Code 2.5 it is shown how this is done for a 2D matrix.

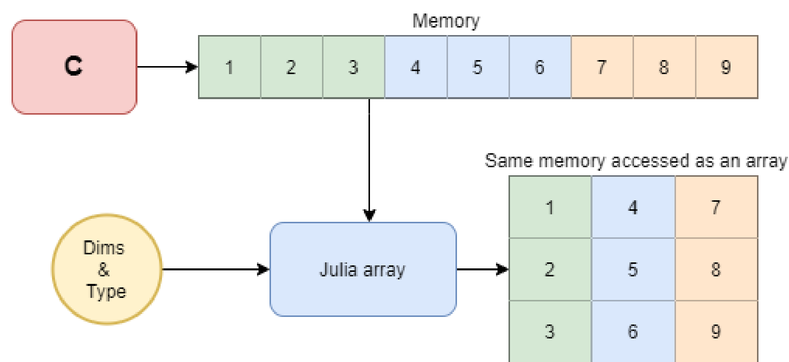


Figure 2.2: Julia arrays that share memory with C can be created by pointing at a shared peace of memory and providing the dimensions of the array. The illustration shows a 2D matrix for simplicity this method is valid for matrices of n dimensions.

Source Code 2.5: Sharing memory

```
// How to share a 2D matrix between C and Julia
// Allocate memory for array and a pointer to it
double *C_point =(double*)malloc(dim1*dim2*sizeof(double));

// Get array dimensions as a Julia tuple
jl_value_t* dimstup = jl_eval_string("(dim1, dim2)");
jl_value_t* jl_complexf64_type = jl_eval_string("Complex{Float64}")


// Get type of array
jl_value_t* arrtype = jl_apply_array_type((jl_value_t*)jl_complexf64_type,2);

// instatiate julia array as the Memory pointed to by C_point
jl_array_t* julia_data = jl_ptr_to_array(arrtype,C_point,dimstup,0);
```

One has to be wary of the fact that Julia stores memory in column-major, unlike C that uses row-major which might lead to indexing errors. Figure 2.3 illustrates the difference in memory storage, which can be managed by transposing the matrices on either end or just index accordingly.

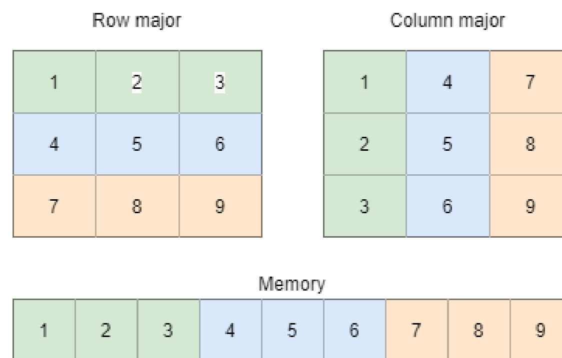


Figure 2.3: Illustration of how memory in arrays are saved in different languages.

3

Radar Signal Processing

In this chapter, some concepts in radio detection and ranging (radar) is introduced. The basic idea behind radar is that an electromagnetic pulse is emitted at the speed of light c from an antenna. The pulse then bounces on a target back to the antenna the range R can the be calculated based on the time t it took for the signal to return $R = tc/2$. The radar equation in Equation (3.1) describes power received by an echo returning from a radar target.

$$P_r = \frac{P_t A_e^2 \sigma}{4\pi^2 \lambda^2 R^4} \tag{3.1}$$

P_r	:	Recieved power	W
P_t	:	Transmitted power	W
A_e	:	Effective aperture (area) of receiving antenna	m^2
σ	:	Radar cross section	m^2
λ	:	Wavelength of signal	m
R	:	Range between target and receiver	m

The equation is central for radar design as it indicates how certain physical parameters will affect the effectiveness of the radar.

3.1 Pulse-Doppler Radar

Pulse doppler radar is a type of radar that can identify moving targets by differences in received frequencies relative to the transmitted frequencies. These differences occurs due to the Doppler effect equation (3.2) and will be proportional to the change in range between target and radar.

$$\Delta f = -2\frac{\dot{R}}{\lambda}, \qquad \lambda = \frac{c}{f} \tag{3.2}$$

Δf	:	Doppler frequency	Hz
f	:	Transmitted frequency	Hz
λ	:	Wavelength of signal	m
$\dot{R}$	:	Change in range with respect to time	m/s
c	:	Speed of light	m/s

Exploiting Doppler is fairly straight forward for a stationary ground based radar, however for the airborne counterparts it gets more complicated as they are in motion.

The motion will lead to Doppler frequencies from both static and dynamic objects and it gets harder to distinguish between actual moving targets and ground clutter.

3.2 Interference

Radars are subjected to various types of interference which is the reason singal processing is necessary to make any sense of sampled data.

3.2.1 Clutter

As a radar emits signals in all different directions one will receive echoes from objects other than the intended target, common such objects are ground, rain and sea. These returning echoes are referred to as clutter.

3.2.2 Jamming

Non civil entities might attempt to disturb your radar by deploying electronic counter measures (ECM) in the form jamming devises that emits a broad band of frequencies to corrupt the data sampled by the radar [7]. Received power from the jamming device is described in Equation (3.3).

$$P_n = \frac{P_j G_j A_e}{4\pi R_j^2} \tag{3.3}$$

P_n	:	Recieved power from the jammer	W
P_j	:	Transmitted power from the jammer	W
A_e	:	Effective aperture (area) of receiving antenna	m^2
G_j	:	Jammer gain	1
R_j	:	Range between jammer and receiver	m

The received power from jamming sources can thus be much larger than that of radar echoes as it is only attenuated by a factor R^{-2} compared to that of the radar echo that is attenuated by a factor R^{-4} see Equation (3.1).

3.2.3 Noise

Noise is interference from any undistinguished source.

3.3 AESA-Radar

Active electronically scanned array (AESA) is a type of phased array antenna that consists of a grid of antenna elements that all transmit and receive signals see Figure 3.1.

Figure 3.1: AESA antenna

With this grid phase shifts are used to create constructive interference in order to listen in a particular direction, this is called beam forming. Returning echoes from the specified direction are amplified and echoes from other directions are reduced. AESA distinguishes itself from other antennas by being able to electronically steer the antenna beam almost instantly compared with mechanically steered antennas that require physical rotation of the antenna which is much slower. An example of the AESA signal processing chain that will be examined for this thesis can be seen in Figure 3.2. The chain consists of digital beam forming, pulse compression, moving target indicator, doppler processing and detection, these processes will be explained in detail in Sections 3.5 to 3.10 .

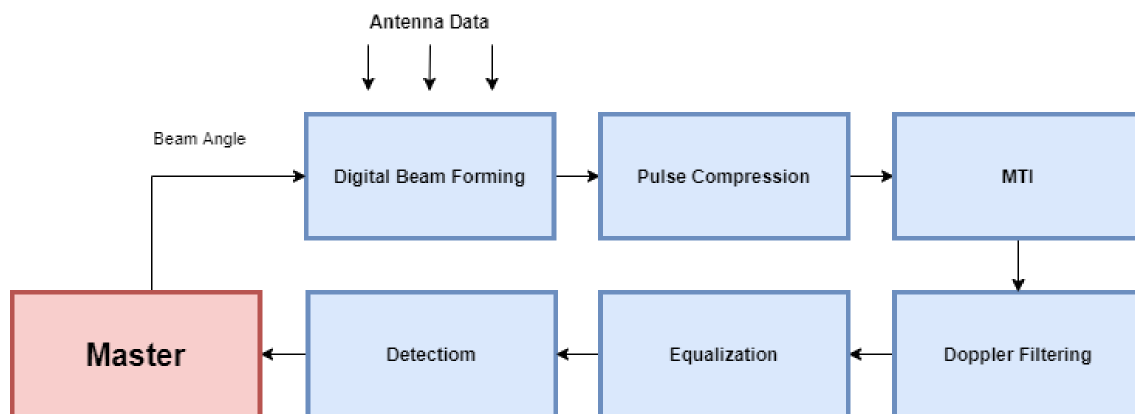


Figure 3.2: Signal processing chain

3.4 Data acquisition

[18, Chapter 3] When a radar transmitter emits an electromagnetic pulse the receivers start to sample data over a fixed amount of time called the pulse repetition interval (PRI). Once the PRI has passed a new pulse is sent out and the process is repeated at a rate reffered to as PRF (Pulse Repetion Frequency) $PRF = 1/PRI$.

The sampled pulses are stacked into a 2D data matrix that constitutes the signal processing input for a single antenna channel see Figure 3.3 this matrix is reffered to as an INTI (Integreation interval). As the AESA radar has several antenna channels several such matrices are created and stacked into a data block see Figure 3.4.

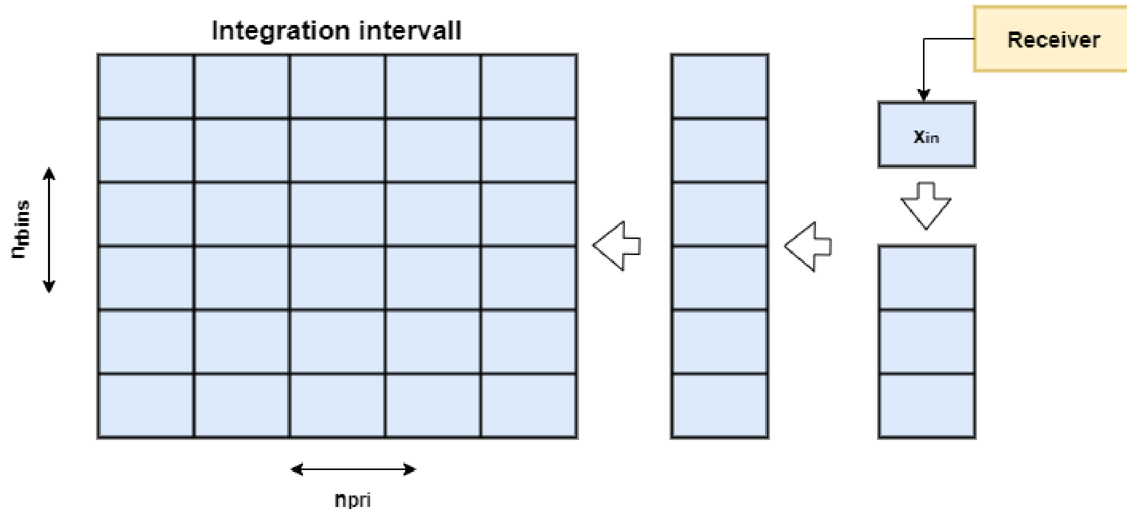


Figure 3.3: Illustration of how data is sampled and arranged into an INTI from a single antenna element

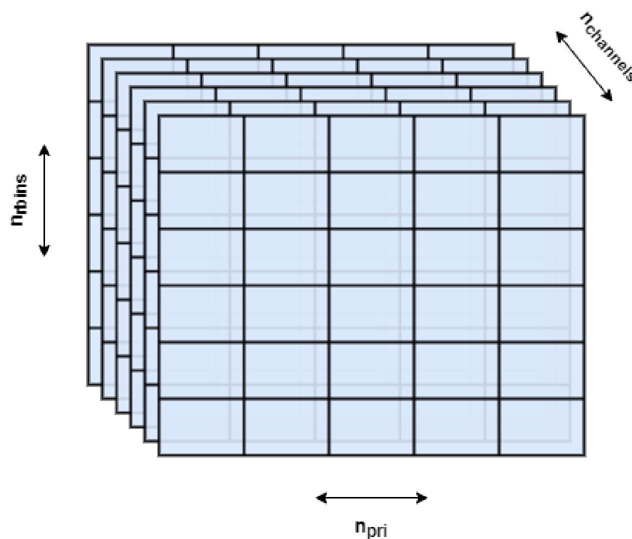


Figure 3.4: INTI's from several antenna channels are stacked up into a block of data

3.5 Digital beamforming

Digital Beamforming (DBF) is the process of synchronizing signals from different antenna channels with phase shifts such that constructive interference creates coherent listening beams in the desired directions [18, Chapter 9]. The signal from the listening beam is calculated in Equation (3.4). Echoes are still perceived from other directions in what is called side-lobes however they are amplified to a lesser extent and thus need to contain a lot of energy in order to result in false detections etc.

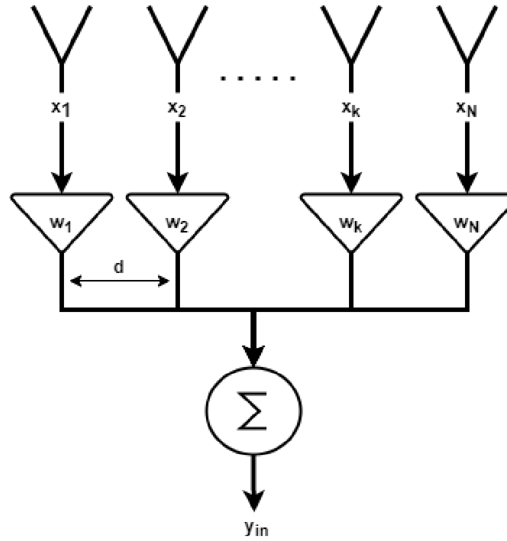


Figure 3.5: Diagram of how the AESA-antenna merges individual element signals into one coherent signal. For simplicity the antenna depicted has a linear configurations of elements which makes it easier to calculate weights for the beam-forming process see Equation (3.5).

$$y = \sum_{k=1}^{n_{el}} x_k w_k, \qquad w_k = c_k e^{j\varphi_k} \tag{3.4}$$

$$\varphi_k = \frac{(k-1)2\pi d \sin(\theta)}{\lambda} \tag{3.5}$$

y	: Calculated signal	1
x_k	: Received signal by individual antenna element k	1
c_k	: Tapering constant that suppresses side lobes	1
φ_k	: Phase shift constant for antenna element k	Rad
θ	: Desired beam angle	Rad
d	: Distance between antenna elements	m
λ	: Wavelength	m

It is possible to calculate multiple beams at the same time by utilizing several weight configurations. This enables the radar to look in several directions at the same time. The algorithm 1 illustrates one possible way to implement the beam-forming process and is to be benchmarked in Julia and Matlab for comparison. In the algorithm, the datablock is sliced into a matrix of dimension $(n_{rbin}, n_{channels})$ for every PRI and multiplied with the weight matrix that and fills up a PRI slice in the resulting data block. As the process reads and writes from unique memory this is valid for parallelization without any risk of a data race. The beamforming process has a computational complexity of $\mathcal{O}(n_{ch} n_{pri} n_{rbin} n_{beams})$.

Algorithm 1: Beam-forming

Data: X is a block with the dimensions $(n_{rbin}, n_{pri}, n_{channels})$ with raw video data, w weight matrix with dimensions $(n_{channels}, n_{beams})$
Result: Post beam forming INTI matrix Y of dimensions $(n_{rbin}, n_{pri}, n_{beams})$
for *i in 1:*n_{pri} **do**
  $X_{pri} = X[:, i, :]$
  $Y[:, i, :] = X_{pri}w$
end
return Y

The number of beams that can be maintained is limited by the computational capacity and memory available on the hardware. For airborne radar, weight and size of the onboard computer is much more limited than for a ground-based radar.

Source Code 3.1: Julia digital beam forming

```
function dbf_t_beams!( X::Array{<:Complex,3},
                       Y::Array{<:Complex,3},
                       w::Array{<:Complex,2} )
    n_pri = size(X,2)
    @threads for i in 1:n_pri
        Y[:,i,:] = X[:, i,:]*w
    end
end
```

Source Code 3.2: Matlab digital beam forming, the squeeze function converts the data slice from a 3D array to a 2D array

```
function Y=dbf_beams(X,w)
    [n_rbins, n_pri, n_channels] = size(X);
    n_beams = size(w,2);
    Y=zeros(n_rbins,n_pri,n_beams);
    for i = 1:n_pri
        Y(:,i,:) = squeeze(X(:, i,:))*w;
    end
end
```

3.6 Pulse Compression

To get good resolution on readings it is desirable to have short signals [18, Chapter 4]. In order to detect targets however enough energy needs to be transmitted and hardware is limited to a finite power output. One way to handle this is to transmit a long signal of linearly increasing frequency see Equation (3.6) . When the signal is received it is compressed to a short signal with a large amplitude and constant frequency by passing it through a matching filter with a time lag that decreases linearly see Equations (3.7) to (3.8). This is called linear frequency modulation (LFM)

and is commonly referred to as chirping in the industry.

$$x(t) = e^{j\pi\left(\frac{f_s - f_0}{T}t^2 + f_0 t\right)}, \quad 0 \le t \le T \tag{3.6}$$

$$h(t) = x^*(T - t), \quad 0 \le t \le T \tag{3.7}$$

$$y(t) = h * x \tag{3.8}$$

x	: LFM waveform	1
h	: Matching filter	1
f_0	: Start frequency	Hz
f_s	: Final frequency	Hz
t	: Time	s
T	: Pulse duration	s
y	: Compressed signal	1

3.7 MTI

A moving target indicator (MTI) filters out moving targets from stationary targets [28, Chapter 2]. It accomplishes this with a high-pass FIR filter that can be chosen with normalized binomial coefficients with a alternating sign for example $\frac{1}{\sqrt{2}}[1, -1]$, $\frac{1}{\sqrt{6}}[1, -2, 1]$, $\frac{1}{\sqrt{20}}[1, -3, 3, -1]$ and so on. In Figure 3.6 a flowchart of the first filter is shown where the difference between consecutive pulses calculated. Consecutive echoes from static objects should then be the same thus the output from the filtering process would be zero meaning that the clutter would be suppressed. As the filter is fairly small it is no point in transforming it into the frequency domain using fft and can thus be kept in time domain. The case presented here requires the radar to be stationary as well. For a moving and typically airborne radar platform further compensations are necessary however that is beyond the scope of this thesis.

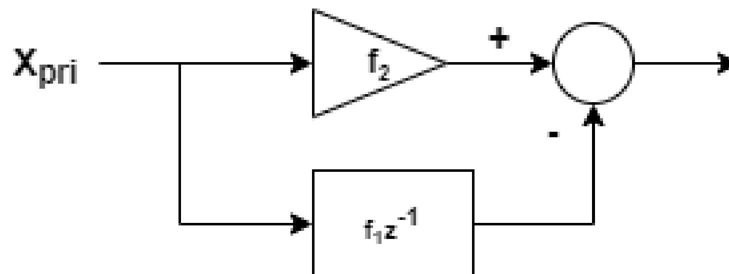


Figure 3.6: MTI with a FIR-filter of size 2

As MTI is a fairly straightforward operation it is used to compare Julia, Matlab and C.

Loop version

Source Code 3.3: MTI loop version Matlab

```
%Matlab
function out = MTI(inti,filter)
    [n_rb, n_pri]  = size(inti);
    out = zeros(n_rb, n_pri - length(filter) + 1);
    for i = 1:n_rb
        for j = 1:n_pri-length(filter)+1
            s = 0;
            for k = 1 : length(filter)
                s = s + filter(k)*inti(i,j+k-1);
            end
            out(i,j) = s;
        end
    end
end
```

Source Code 3.4: MTI loop version Julia

```
#Julia
function MTI!(inti,filter,out)
    n_rb, n_pri  = size(inti)
    #@threads for i = 1:n_rb
    for i = 1:n_rb
        for j = 1:n_pri-length(filter)+1
            s = 0.0
            for  (k,f) in enumerate(filter)
                s += f*inti[i,j+k-1]
            end
            out[i,j]= s
        end
    end
end
```

Source Code 3.5: MTI loop version C

```
// C
    for (l = 0; l < iterations; l++) {
        for (i = 0; i < dim1; i++) {
            for (j = 0; j < dim2+1-filter_length; j++) {
                double s = 0.0;
                for (k=0; k<2;k++)
                {
                    s += filter[k]*inti[i][j+k-1];
                }
```

```
                out[i][j] = s;
            }
        }
    }
```

Vectorised version

To be fair a vectorised version is also tested where Matlab is expected to perform better as it may utilize underlying and highly optimized linear algebra packages.

Source Code 3.6: MTI vectorised version Matlab

```
%Matlab
function out = MTI_mat(inti,filter)
    [n_rb, n_pri]  = size(inti);
    out = zeros(n_rb, n_pri-length(filter)+1);
    for k = 1:length(filter)
        out = out + filter(k)*inti(:,k:end+k-length(filter));
    end
end
```

Source Code 3.7: MTI vectorised version Julia

```
#Julia
function MTI_mat!(inti,filter,out)
    f_len = length(filter)
    for  (k,f) in enumerate(filter)
        out .+= f*inti[:,k:end+k-f_len]
    end
end
```

3.8 Doppler Processing

In the Doppler processing step the contents of every range bin is first multiplied with a window function to suppress side-lobes and is then subjected to an FFT operation for every bin that transforms the data in the data matrix such that the energy is divided into a range bin and frequency bin.

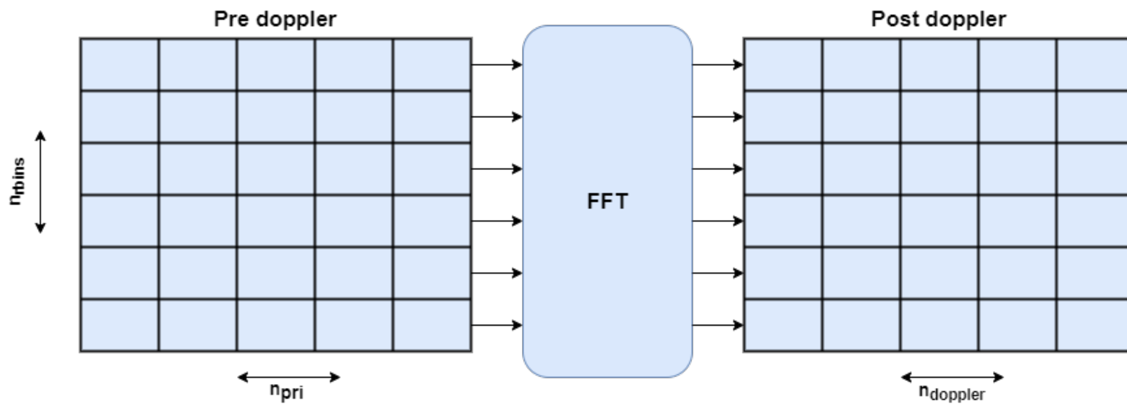


Figure 3.7: Doppler processing

3.9 Equalization

The signal processing performed to remove clutter and move it to the frequency domain will leave the signal distorted. The causes of this distortion is the convolution performed during the MTI operation and the window function applied during the Doppler processing. To compensate for this equalization is performed.

3.10 Detection

A target is detected when a sufficient amount of energy is accumulated in a cell in the data matrix. To avoid false detections a detection scheme called constant false alarm rate (CFAR) is employed which applies a normalised threshold that hopefully filters out false detections and only leaves real targets. Tuning these thresholds are tricky as if they are set to high real targets might be filtered away and if they are too low one might let through several false targets. With energy returns organized in range and frequency bins, it is possible to determine both range and relative speed to the radar. Detection will not be further explored beyond this which is included to get perspective as to what the processed data will be used for.

3.11 STAP

As was described in section 3.2.2 interference from jamming is an issue for deployed radars [18, Chapter 9]. There are methods to mitigate the effectiveness of jamming devices as long as they are not broadcasting from the direction of the main lobe. The basic idea is to use a sampled block of radar data to calculate tapering weights for the beamforming process such that side-lobes in the direction and frequency of high interference are zero and thus cancelled out. This is called side-lobe cancellation (SLC) and is a space time adaptive algorithm process (STAP). The SLC will be used as a representative heavy workload to compare C++, Julia and Matlab in terms of execution time and terms of code complexity.
Unlike the SP-chain in Figure 3.2 the slc beamforming process isn't done untill after

pulse compression and doppler filtering Figure 3.8. This means that the computations of these steps increase proportionally with the amount of antenna channels.

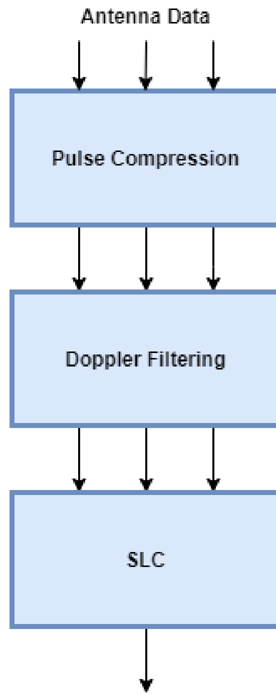


Figure 3.8: Post doppler SLC

The input to the SLC in Algorithm 2 is a sampled data block X of dimension $(n_{rbin}, n_{dopp}, n_{ch})$ and a steering vector s of length n_{ch} that contains phase-shift constants to achieve constructive interference in desired direction. The block is sliced for every Doppler channel into matrix X_{dopp} and an esitimation of a covariance matrix is calculated $\bar{R}_{dopp} = X_{dopp}X_{dopp}^H$. The inverse of the the covariance matrix is then multiplied with the steering vector to produce a weight vector $w = \bar{R}_{dopp}^{-1}s$. w can then be used for beam-forming of the corresponding frequency channel.

Algorithm 2: SLC

Data: X is a block with the dimensions $(n_{rbin}, n_{dopp}, n_{ch})$ with raw video data,
s is a steering vector of length $n_{channels}$
Result: Post beam forming INTI matrix Y of dimension (n_{rbin}, n_{dopp})
for *i in 1:*n_{dopp} **do**
  X_{dopp} = slice of X in doppler channel
  $R = X_{dopp}X_{dopp}^H$
  $w = R^{-1}s$
  $w = w/w[1]$
  $Y[:, i] = X_{dopp}^T w$
end
return Y

In Source Code 3.8 one can see the Julia implementation of Algorithm 2. Notable here is the use of view which eliminates the need to make a copy the data slice and thus keeps allocations down.

Source Code 3.8: Julia SLC

```
# Julia
function slc!(X,Y,s)
    n_doppler= size(X,2)
    @threads for loop = 1:n_doppler
        #X_dopp = X[:,loop,:]  is also valid
        #but view is cheaper in terms of allocations
        X_dopp = transpose(view(X,:,loop,:))
        R = X_dopp*X_dopp'
        w = R\s
        w = w/w[1]
        Y[:,loop] = transpose(X_dopp)*w
    end
end
```

In Source Code 3.9 one can see the Matlab implementation of Algorithm 2 which closely resembles the Julia counterpart with some noteworthy differences. It is not possible to mutate memory with Matlab functions it is thus necessary to allocate new memory and return the output. Second Matlab has to allocate a new array for the data slice X_dopp as it does not have a counterpart to view.

Source Code 3.9: Matlab SLC

```
% Matlab
function Y = slc(X,Y,s)
    N_doppler = size(X,2);
    for loop = 1:N_doppler
        X_dopp = squeeze(X(:,loop,:)).';
        R = X_dopp * X_dopp';
        w = R\s;
        w = w/w(1);
        Y(:,loop) =  X_dopp.' *w;
    end
end
```

In Source Code 3.10 one can see the C++ implementation of Algorithm 2. Where the previous examples are fairly straight forward to implement this case requires a significant amount of work even though it utilizes the third party linear algebra package Eigen [5] which is considered to be expressive and easy to use for C++.

Source Code 3.10: C++ SLC

```
//C++
int slc(MatrixXcd &Y, Eigen::Tensor<std::complex<double>,3> &data)
{

  int n_rbins   = data.dimension(0);
  int n_doppler = data.dimension(1);
```

```
  int n_channel = data.dimension(2);
  int i;

  MatrixXcd X_view;
  MatrixXcd R;
  MatrixXcd w;

  VectorXcd s = VectorXcd::Zero(n_channel);
  s[0] = {1.0,0};

#pragma omp parallel for private(i, X_view, R, w)
  for(i=0;i<n_doppler;i++)
  {

    std::array<long,3> offset = {0,i,0};        //Starting point
    std::array<long,3> extent = {n_rbins,1,n_channel};    //Finish point
    std::array<long,2> shape2 = {n_rbins,n_channel}; //Shape of  tensor (matrix)

    MatrixXcd C = Tensor_to_Matrix((Eigen::Tensor<std::complex<double>, 2>)data.sli
    X_view = C.transpose();

    R = X_view * X_view.adjoint();

    w = R.lu().solve(s);
    w = w / w(0);

    Y.col(i) = X_view.transpose()*w;

  }

  return 0;
}
```

4
Results and Discussion

In this chapter results for some test that illustrates the characteristics of Julia will be presented as well as a discussion about the results. Performance of processes in previous chapters will only be included individually if they are compared to implementations in another language. Julia and C tests are conducted from a virtual machine using an ubuntu 20.04 operating system. Matlab tests are conducted on Windows 10 operating system. Tests, where Julia is compared with Matlab, are done with the type Float64 using 64 bits as Matlab by default uses the type double which also uses 64 bits. Note however that it is more prudent to use Float32 as that only requires the use of half as much memory and is of sufficiently high precision, thus this is used when Julia is not compared with Matlab.

4.1 Beamforming

In Table 4.1 runtimes for a beam-forming process utilizing the same algorithm is compared. To receive these results it is necessary to actively specify the amount of threads used by Matlab to 1 with "maxNumCompThreads(1)".

Table 4.1: Mean runtime results for a beam-forming computation of dimensions (256,256,32) as matlab stores complex numbers using 16 bytes, Complex{Float64} is used for Julia.

Julia	Matlab
9.056 ms	7.936 ms

In Figure 4.1 it can be seen how larger precision variables can adversely affect runtimes due to the larger amount of memory allocated. Beam forming processes a very large amount of data and the vectorised implementation is not free from allocations which leads to a high cost when using higher precision variables which can be seen in the figure.

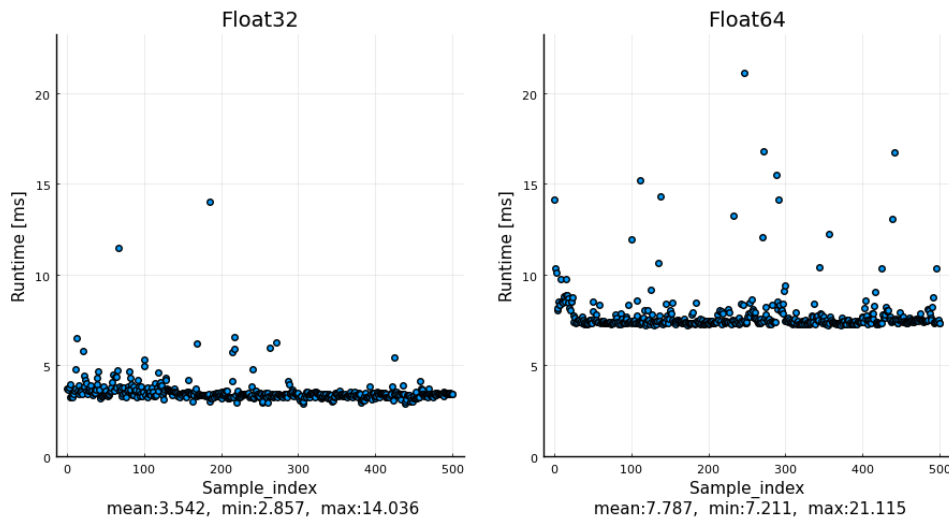


Figure 4.1: Threaded beamforming in Julia performed with data types Complex{Float32} and Complex{Float64} using 4 threads. Input dims: (256,256,32) output dims:(256,256,4)

4.1.1 Discussion

In the beam-forming process there are a lot of matrix multiplications something that Matlab is known to do very well as was shown in [16]. Thus the results in Table 4.1 where Matlab performs slightly better is not surprising. Vector operations in Julia is seldom free from allocations which becomes expensive with large arrays and high precision types. The result in Figure 4.1 illustrates the cost of higher precision types for large allocating functions.

4.2 MTI

Table 4.2 contains the result from the different MTI implementations presented in Section 3.7. In the tests a normalized 2 element filter is used $[-1/\sqrt{2}, 1/\sqrt{2}]$.

Table 4.2: Mean runtimes MTI dims: (512,512)

C	Matlab scalar	Matlab vectorized	Julia scalar	Julia vectorized
1.73ms	31.3ms	4.1ms	1.64ms	1.92ms

4.2.1 Discussion

The test results in Table 4.2 shows that Julia is able optimize scalar operations to a level very similar to the C counterpart where as Matlab fails to perform for the scalar case. This gives Julia an edge as one may get a better indication of how an algorithm will perform in a lower level implementation already in a prototyping phase. The vectoriced case is more equal however Julia still outperforms Matlab and delivers approximately twice the speed. What is particularly noteworthy is

that Julia can actually optimize these sort of operations such that they can even beat imported highly optimized libraries written in C.

4.3 Embedding

In this section, the results of employing the embedding on an SP-chain is presented see Source code A.1 and A.2 for the embedded Julia code and the C code used respectively.

4.3.1 Serial test

In Figure 4.2 it can be seen how a Julia signal processing chain performs embedded into a C program relative directly from Julia. In this test run the embedded version was both faster and more deterministic with lower variance than the stand-alone Julia counterpart. This is not necessarily always the case the same test on another computer resulted in stand-alone Julia being slightly faster while still being less deterministic. This however proves that there is no great penalty in embedding a Julia module.

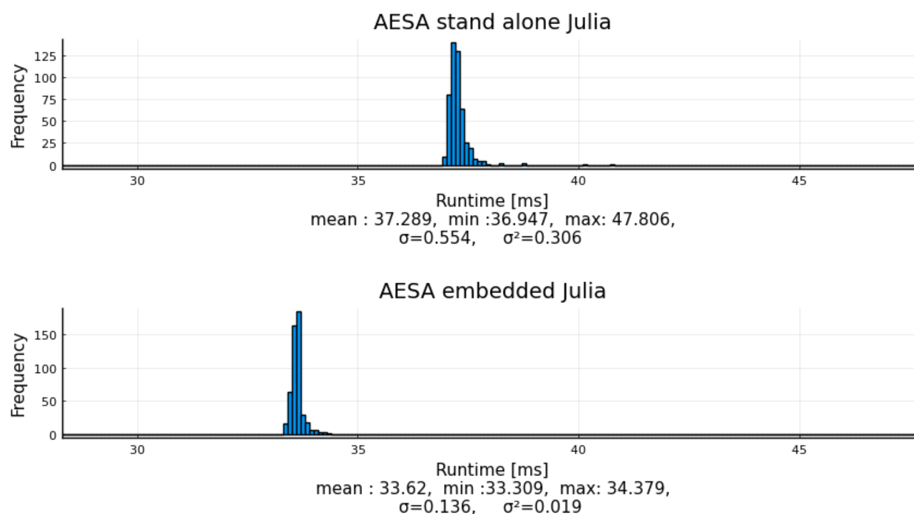


Figure 4.2: Julia SP-chain embedded into C compared to the same SP-chain performed from standalone Julia

4.3.2 Threaded test

In Figure 4.3 the same test was conducted while threads were enabled. In this test, it can be seen that the stand-alone Julia version is accelerated whereas the performance of the embedded version deteriorates significantly while threads are enabled. Similar issues occur when the amount of BLAS threads are set higher than one which they are by default if Julia uses OpenBLAS.

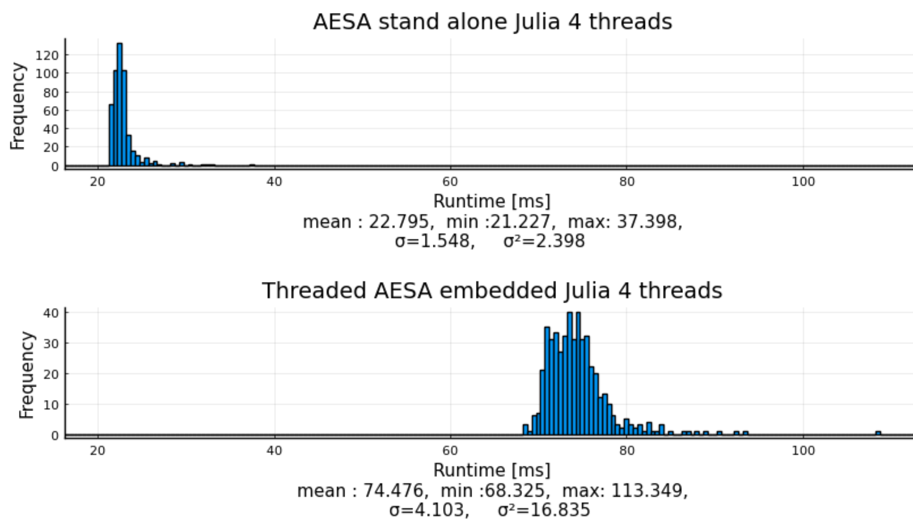


Figure 4.3: Threaded Julia SP-chain embedded into C compared to the same SP-chain performed from standalone Julia

4.3.3 Discussion

From these results, it is possible to conclude that Julia is indeed viable to embed while still retaining the performance desired for serial execution. However, enabling threads for the embedded threaded functions be they self-made or part of a BLAS library is not viable with the proposed method. For this reason, it is also necessary to actively change the amount of BLAS threads otherwise it will appear that embedding Julia comes with a lot of overhead when in reality it does not.

4.4 SLC test

In this section implementations of Algorithm 2 will be compared between Matlab, Julia and C++. The comparison is made for two Linear Algebra vendors OpenBLAS which is used as default by Julia and MKL which is used as default by Matlab. The original tests were done on an AMD processor where the usage of MKL didn't result in any performance gain and in Julia's case even had a decremental effect. According to some sources [20] AMD processors are sub-optimal for MKL as it is a software developed by Intel and thus optimized for Intel processors. For this reason, the tests were also done on an Intel processor to see the difference between MKL and OpenBLAS. Note that the specifications of the Intel computer exceed that of the AMD computer.

4.4.1 Runtimes with AMD

In Table 4.3 no conclusive difference between MKL and OpenBLAS can be seen for C++. For Julia MKL performed somewhat worse than OpenBLAS. Matlab was competitive in this test and seems to utilize MKL more effectively than the others.

Table 4.3: Mean runtimes of SLC with dims (128,256,16) using AMD Ryzen 9 4900HS with Radeon Graphics 3.00 GHz 8 cores

Threads	C++ OpenBLAS	C++ MKL	Julia OpenBLAS	Julia MKL	Matlab MKL
1	10143μs	10174μs	12957μs	13928μs	6536 μs
2	5101μs	5101μs	5199μs	7396μs	-
4	2726μs	2665μs	2752μs	3862μs	-
8	1602μs	1585 μs	1672μs	2611μs	-

4.4.2 Runtimes with INTEL

Table 4.4: Mean runtimes of SLC with dims (128,256,16) using Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz 16 cores

Threads	C++ OpenBLAS	C++ MKL	Julia OpenBLAS	Julia MKL
1	7285μs	3238.7μs	8980μs	8951μs
4	-	845.9μs	2481μs	2348μs
8	-	460.2 μs	1568μs	1943 μs

In Table 4.4 time comparisons of the SLC processes are presented for C++ and Julia utilizing different BLAS libraries. It can be seen that Julia is fairly close to C++ with OpenBLAS but fails get the same boost from MKL as C++ does for this case. This test was done on an Intel processor which is required to get the full benefit from MKL.

4.4.3 Discussion

From the results here we can see that Julia fails to achieve the same performance boost from MKL that C++ does. As radar signal processing mostly consists of large linear algebra operations this is unfortunate. With the release of Julia 1.7 better support for arbitrary BLAS libraries is expected and hopefully improves on these issues.

4.5 Garbage Collection

4.5.1 Overhead of calls to GC

Using the benchmark functions provided by the package *BenchmarkTools* the general overhead of manual calls to garbage collection was estimated for a fresh Julia session. The result can be seen in Table 4.5

Table 4.5: Overhead for calls to garbage collection estimated by bench-marking the functions without any garbage, this may differ between different machines but it gives an indication of relative magnitude between full and young sweeps.

GC.gc()	GC.gc(false)
43 ms	47 μs

4.5.2 Managed GC vs unmanaged GC

Figure 4.4 shows how runtimes in Julia often get a bimodal distribution if GC is not managed. It can further be seen that by manually calling the garbage collection for a sweep of young objects during every cycle it is possible to make it more deterministic at the cost of some overhead. While the improvements are noticeable there are still some outlier runtimes which is unfortunate.

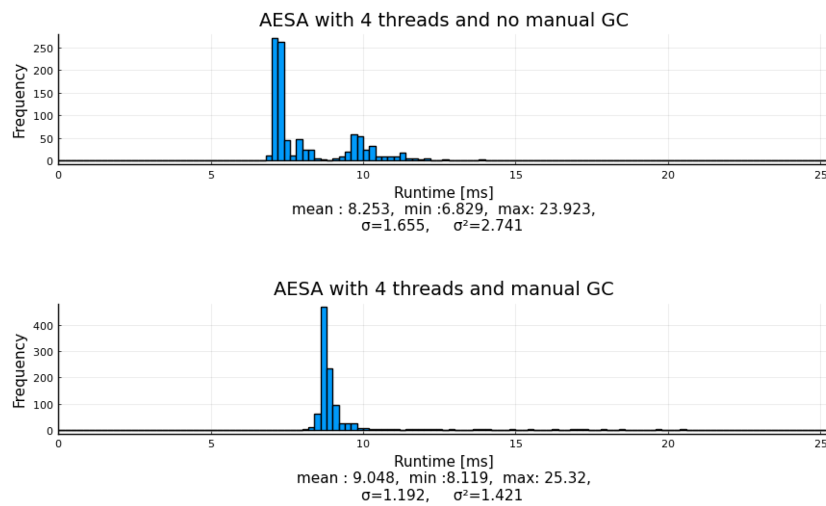


Figure 4.4: Runtime distribution for consecutive aesa signal processing cycles in Julia when garbage collection is left unmanaged and when GC is called for every cycle. Dimensions used are $(n_{rb} = 128, n_{pri} = 256, n_{channels} = 16, n_{beams} = 4)$. Data type used is Complex{Float32}

Figure 4.5 illustrated the effects of an unmanaged GC compared to that of a managed one in a test where a threaded SLC function ran in an iterative manner.

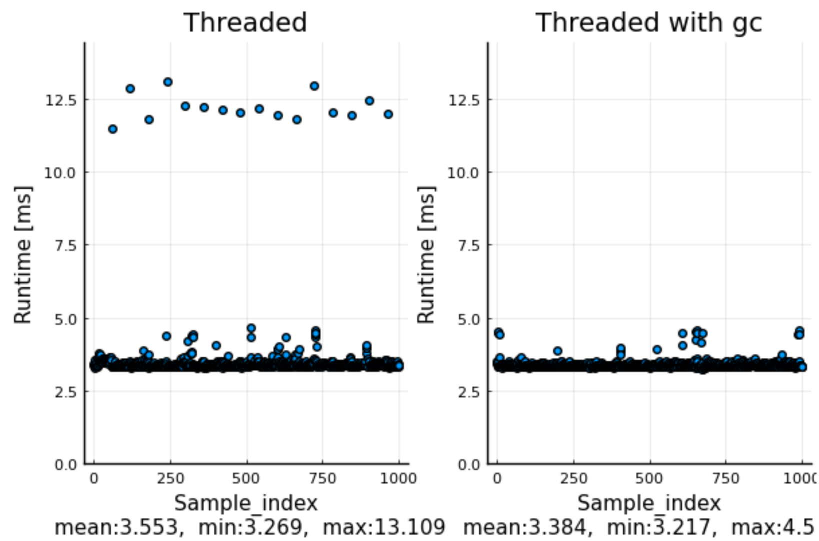


Figure 4.5: The figure illustrates two sets of runtime samples for a side-lobe cancellation process. In the plot to the left garbage collection is not being managed and it can be seen that a few runtimes have a large offset from the average runtime. In the right plot, garbage collection for young objects is done for every sample, this measure appears to remove the large outliers observed in the prior plot.

4.5.3 Discussion

The garbage collection in Julia is a very convenient feature and good for productivity at the development stage as the programmer doesn't have to deal with memory management. If one has real-time requirements it is possible to manually call GC of young objects for every processing cycle. The possible overhead of young GC calls at suitable places is fairly small and almost negligible if the cycles are sufficiently. Full GC should not be called during any time-sensitive executions as it has considerable overhead several orders of magnitude greater than the young sweep. As it is only of interest to remove garbage allocated in the previous cycle this is no limitation for the case viewed. Calling GC from multiple threads is ill-considered as all threads stop during the sweep and interrupt each other.

The best way to limit issues with GC is to minimize the amount of allocations such that sweeps are not necessary. The suggested way to deal with these issues is to pre-allocate as much memory as possible and write functions that mutates existing memory rather than allocating new memory. Avoiding allocations is easier for scalar operations on immutable objects as there is no size ambiguity, however for vectorised code it is harder to avoid allocations. In the paper [15] the effects of garbage collection is mitigated by utilizing Static Arrays[10] which is an immutable stack allocated data structure. The paper is concerned with control of robotics and works with rather small arrays making the use of static arrays possible. According to the static array documentation they should only be used for matrices ranging up to about 100 elements, for this reason it is not possible to use such when working with the massive amount of data that comes with AESA.

4.6 Compared to Matlab

If Julia is used at SAAB it will to some extent replace Matlab and it is, therefore, prudent to compare the two.
The major strength of Julia is that its compiler can optimize user-defined functions such that they run very efficiently without the need to vectorise. In Section 4.2 a clear example of how Julia exceed Matlab in this regard was shown.
The optional types of Julia gives the user a clear indication of how values are actually stored and enable the developer to control values etc. Types make documentation much clearer compared with Matlab where it easily gets ambiguous if input arguments are scalar, arrays etc. This way of type specification also enables Julia to make use of multiple dispatch that brings great flexibility and cleaner more intuitive libraries as the same function name can be used for methods with different configurations of arguments and types.
Julia is much more accessible than Matlab in terms of cost and size. Costs because Julia itself is free, most packages are free and usually the ones that are not only costs money for commercial use. Matlab on the other hand costs a fair bit of money just to unlock basic features, unlocking further packages can then be very expensive. Julia is small the compressed binary folder is only about 120 MB and uncompressed it is less than 500 MB. What takes up size in Julia is packages and one can be restrictive in which packages one downloads. Matlab is comparatively large and not always practical to install on all devices or setup on virtual machines etc.
The downsides of Julia are mostly related to the fact that Julia is a young language maintained by a relatively small team. Because of this books and other resources are very limited. As Julia is still in an evolving stage with constant changes learning material quickly gets outdated. It is commonly said for programming in general that every problem you encounter someone else has encountered before you and a solution is only a google search away. This is however not always the case for Julia as the community is relatively small.
Julia does not have all features of Matlab such as Simulink for block programming which is a very powerful for prototyping
Plotting in Julia is at the time of writing not ideal where Julia itself only provides a front-end [9] and loads back-ends from third partys such as Pyplot[25]. Loading a back-end from third party takes some time and one can expect to wait approximately 10 seconds to produce the fist plot during a Julia session, subsequent plots would however be produced swiftly. It has also been noted that general performance deteriorated to some degree when a back-end is loaded. The front-end is however very expressive and one can produce good looking plots, though it takes a while.

4.7 Compared C/C++

As Julia is a higher-level language than C/C++ it is much easier learn and operate. The large difference between Julia and C/C++ is that it is not statically compiled this comes with both benefits and drawbacks. As Julia compiles during runtime it is possible to write non-type specific code and instead infer types during runtime giving

it a larger range and more intuitive syntax. The obvious downside is that it has to compile during runtime which can make Julia perform somewhat slow before it gets started. With the use of system images that are described in Appendix A.2 one can to some degree combat these issues however that is far from a perfect solution.
C/C++ has much stiffer constraints as everything must be specified at compile time making it harder to code and is thus to some extent less modular than Julia. Whereas Julia's garbage collection is a nice feature in a prototyping phase it is hard to justify it for safety-critical systems as it is somewhat unpredictable. For this reason, one is likely better off with manual memory management in C/C++ as that is more predictable. If one wants to produce a program that runs instantly and consistently every time one is likely better off with a statically compiled language such as C/C++. With this being said Julia is still very good for prototyping purposes which is not something one does in C/C++ today anyway.

4.8 Two language problem

Julia is fast and certainly solves the two language problem for many users in academia for people who want to run fast simulations with self-written algorithms for example [15]. This is made possible by the JIT compiler that dynamically builds very efficient functions with short execution times after the initial run.
The two language problem will not be eliminated by introducing Julia at SAAB it would essentially fill a similar role as Matlab does today with the option to embed models in a proper C/C++ framework for testing on hardware during a prototyping phase. The custom of keeping development in two languages is not necessarily bad either as focus on performance too early can have a negative impact on productivity. With implementations in two languages, meaningless micro-optimization can be avoided during development as that is typically handled by someone else at a later stage.
Implementation in two languages is also valuable for validation where one can be used to find bugs and verify the other. As it is possible to embed Julia it should be possible to create automated unit tests in other languages with Julia as a reference model.

5

Conclusion

In conclusion, Julia is definitely a useful language that could be used effectively at SAAB mainly for prototyping and validation. As has been shown it is also an effective tool to embedded high-level functionality into a lover-level language even though it is limited to serial execution. Julia is an excellent sandbox environment to experiment with parallel code with tools such as multi-threading. With Julias JIT compiler and type system, it is possible to optimize operations in a very efficient manner making it very competitive in terms of speed. However, as it is possible and often more convenient to vectorise most of the operations in radar signal processing one should probably not expect too much in terms of a speed increase relative to languages such as Matlab or Python. At this time it is not advisable to use Julia for any safety-critical systems with strict real-time dependencies as little concern for real-time properties has been made in the design of Julia.

6
Future Work

Julia source code is at present completely unprotected and stored in a script file for anyone to read. To implement Julia in any commercial product it is necessary to protect sensitive source code with some sort of encryption. Further investigation into cross compilation and offline deployment on target platform is required, the package *BinaryBuilder* [2] is worth investigating for that purpose.
It should be monitored if the issues regarding MKL raised in this thesis are resolved with Julia v1.7.x that promises substantial improvements for BLAS libraries.

Bibliography

[1] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. *Julia: A Fast Dynamic Language for Technical Computing*. 2012. arXiv: 1209.5145 [cs.PL].

[2] *BinaryBuilder*. URL: https://binarybuilder.org/ (visited on 05/28/2021).

[3] Jiahao Chen and Jarrett Revels. "Robust benchmarking in noisy environments". In: *CoRR* abs/1608.04295 (2016). arXiv: 1608.04295. URL: http://arxiv.org/abs/1608.04295.

[4] *Contents · DSP.jl*. URL: https://docs.juliadsp.org/stable/contents/ (visited on 05/27/2021).

[5] *Eigen home page*. en. URL: https://eigen.tuxfamily.org/index.php?title=Main_Page (visited on 08/06/2021).

[6] *Embedding Julia · The Julia Language*. URL: https://docs.julialang.org/en/v1/manual/embedding/ (visited on 04/29/2021).

[7] Alfonso Farina. *Antenna-based signal processing techniques for radar systems.* Artech House antenna library. Artech House, 1992. ISBN: 0890063966.

[8] *Home · PackageCompiler*. URL: https://julialang.github.io/PackageCompiler.jl/dev/ (visited on 05/11/2021).

[9] *Home · Plots*. URL: http://docs.juliaplots.org/latest/ (visited on 08/14/2021).

[10] *Home · StaticArrays.jl*. URL: https://juliaarrays.github.io/StaticArrays.jl/stable/ (visited on 08/08/2021).

[11] *JuliaCI/BenchmarkTools.jl*. original-date: 2016-02-23T20:54:02Z. May 2021. URL: https://github.com/JuliaCI/BenchmarkTools.jl (visited on 05/06/2021).

[12] *JuliaMath/FFTW.jl*. original-date: 2017-05-23T19:34:29Z. May 2021. URL: https://github.com/JuliaMath/FFTW.jl (visited on 05/27/2021).

[13] Stefan Karpinski. *Detail about Julia's Garbage Collector*. en. URL: https://discourse.julialang.org/t/details-about-julias-garbage-collector-reference-counting/18021/3 (visited on 05/16/2021).

[14] Stefan Karpinski, Viral Shah, Alan Edelman, and Jeff Bezanson. *Julia 1.6 Highlights*. en. URL: https://julialang.org/blog/2021/03/julia-1.6-highlights/#compile_time_percentage (visited on 05/06/2021).

[15] Twan Koolen and Robin Deits. "Julia for robotics: simulation and real-time control in a high-level programming language". In: May 2019. DOI: 10.1109/ICRA.2019.8793875.

[16] Jules Kouatchou. *NASA Modeling Guru: Basic Comparison of Python, Julia, Matlab, IDL and Java (2019 Edition)*. URL: https://modelingguru.nasa.gov/docs/DOC-2783 (visited on 06/18/2021).

[17] *Lecture 11 Slides | Performance Engineering of Software Systems | Electrical Engineering and Computer Science | MIT OpenCourseWare*. en. URL: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/lecture-slides/ (visited on 08/04/2021).

[18] Ph D. Mark A. Richards. *Fundamentals of Radar Signal Processing, Second Edition*. en. McGraw-Hill Education, 2014. ISBN: 978-0-07-179832-7.

[19] *Memory Management Glossary: G — Memory Management Reference 4.0 documentation*. URL: https://www.memorymanagement.org/glossary/g.html (visited on 05/16/2021).

[20] *Mingru Yang - MKL has bad performance on an AMD cpu*. sv. URL: https://sites.google.com/a/uci.edu/mingru-yang/programming/mkl-has-bad-performance-on-an-amd-cpu (visited on 06/18/2021).

[21] *Multi-Threading · The Julia Language*. URL: https://docs.julialang.org/en/v1/manual/multi-threading/ (visited on 05/11/2021).

[22] *Numba: A High Performance Python Compiler*. URL: http://numba.pydata.org/ (visited on 05/30/2021).

[23] *partr thread support by stevengj · Pull Request #105 · JuliaMath/FFTW.jl*. en. URL: https://github.com/JuliaMath/FFTW.jl/pull/105 (visited on 06/01/2021).

[24] *Pkg.jl*. URL: https://pkgdocs.julialang.org/v1/ (visited on 05/11/2021).

[25] *Pyplot tutorial — Matplotlib 3.4.3 documentation*. URL: https://matplotlib.org/stable/tutorials/introductory/pyplot.html (visited on 08/14/2021).

[26] *Race Condition vs. Data Race – Embedded in Academia*. URL: https://blog.regehr.org/archives/490 (visited on 06/04/2021).

[27] Christopher Rackauckas. *mitmath 18337 lecture 5*. en. URL: https://github.com/mitmath/18337 (visited on 05/01/2021).

[28] Merrill I. Skolnik. *Radar Handbook, Third Edition*. en. McGraw-Hill Education, 2008. ISBN: 978-0-07-148547-0.

[29] *Standard Modules · The Julia Language*. URL: https://docs.julialang.org/en/v1/manual/modules/#Standard-modules (visited on 05/30/2021).

[30] *The LLVM Compiler Infrastructure Project*. URL: https://llvm.org/ (visited on 05/26/2021).

[31] *Types · The Julia Language*. URL: https://docs.julialang.org/en/v1/manual/types/ (visited on 06/03/2021).

A

Appendix

A.1 Package environments

In this section the basics of the package manager [24] and project environments in Julia are explained.
The Julia package manager can be accessed in two ways either interactively through the Julia REPL by typing "]" to access pkg mode, alternatively, one can import the package manager and access it within a script.

```
# pkg mode in REPL
julia>]
(@CurrentProject) pkg> <function> <argument>

# access Pkg inline
import Pkg
Pkg.<function>("<argument>")
```

It doesn't matter which of the above alternatives are used, in the upcoming examples, the latter will be used. To get started in a new environment simply run the activate command in the working directory, the environment will not contain any packages upon creation. Packages are simply added with the add command which will add information about the package to a file called Project.toml, if the package does not already exist on the computer it will also be downloaded. In addition to the Project.toml there is a file called Manifest.toml that charts all dependencies including indirect ones. The environment only consists of these two files and is thus very cheap in terms of memory.

```
# How to create a Project environment
import Pkg
# activate an environment in the current directory
Pkg.activate("CurrentProject")
Pkg.add("SomePackage") #adds a package to the working environment
#information about which packages are added is
#stored in a file named Project.toml

# Update manifest.toml with new dependencies
Pkg.resolve()
```

If one wants to start an existing project environment on a new computer it is possible

to load the environments dependencies from information contained in Project.toml that specifies direct dependencies. If the project also contains the Manifest.toml file it is possible to replicate exactly the same environment including indirect dependencies as those on the computer where the file was created. It is common practice to emit Manifest.toml when working on another machine as it is usually better to have the latest versions of packages, it is therefore included in the default Julia .gitignore. It is however useful when repeatability is desirable.

```
# How to instantiate a Project environment
import Pkg
# activates the environment specified project in Project.toml
Pkg.activate("CurrentProject")

# Download and sets up all dependencies specified in the Project.toml
#and Manifest.toml. If no Manifest.toml exists one will be generated.
Pkg.instantiate()
```

A.2 System image

Julia compiles functions with the type of arguments passed the first time it is called and then keeps them in memory. To get a shorter startup time one can save a so-called System image that contains the packages that are currently being used and the compiled functions which will significantly reduce startup time. As the system image is essentially a snapshot of the state when it was made packages will be constrained to that version even if they are updated in the project thus the system image must be recompiled if a certain package version is needed. System images are dependent on the particular platform you are running on and it is typically not viable to share system images between platforms. System images can be created by the package "PackageCompiler"[8]. When a system image is created it runs a representative script where functions are compiled for the argument types that are being used for that session i.e if the example takes integer arguments it will not compile the function for Float arguments as that would lead to infinitely large builds. If a float argument would be passed to the function it will just compile as normal during runtime. In order to create a system image a C-compiler must be installed.

```
#Create systemimage by running command bellow
in repl within desired environment:
using PackageCompiler
PackageCompiler.create_sysimage(:ModuleName;
    sysimage_path="file_path",
            precompile_execution_file="precompile_file.jl")
```

A.3 Code

A.3.1 AESA Julia

Source Code A.1: aesa.jl

```
using JLD2
using MPDBasic
using Base.Threads
using FFTW
using LinearAlgebra
using BenchmarkTools
using Statistics

function dbf_t_beams!(X::Array{<:Complex,3},
                    Y::Array{<:Complex,3},w::Array{<:Complex,2} )
    n_pri = size(X,2)
    @threads for i in 1:n_pri
        Y[:,i,:] = X[:, i,:]*w
    end
end
function dbf_t_beams2!(X::Array{<:Complex,3},
                        Y::Array{<:Complex,3},w::Array{<:Complex,2} )
    # Used to verify the other one this is slow
    n_rb,n_pri,n_ch = size(X)
    n_beams = size(Y,3)
    @threads for i in 1:n_rb
        for j in 1:n_pri
            for b in 1:n_beams
                s = 0
                for k in 1:n_ch
                    s += X[i,j,k]*w[k,b]
                end
                Y[i,j,b] = s
            end
        end
    end
end
function pulse_compression!(inti:: Array{<:Complex,3},
    pulse  :: Array{<:Complex,1},
    window :: Array{<:Real,1}
    )

    number_of_range_bins = size(inti,1)
    number_of_pri = size(inti,2)
    n_beams = size(inti,3)
```

```
    N_pulse = length(pulse)

    Nfft = nextpow(2, N_pulse + number_of_range_bins - 1)

    matched_filter = conj.(pulse[end:-1:1]) # reverse and conjugate of pulse
    matched_filter = matched_filter.*window

    #awkward manual zeropadding...
    matched_filter_zp = [matched_filter; fill(0+0im,Nfft-N_pulse)]
    matched_filter_rep = repeat(matched_filter_zp, outer = (1,number_of_pri))


    @threads for b in 1:n_beams
        inti_zp = [inti[:,:,b]; fill(0+0im,
        Nfft-number_of_range_bins, number_of_pri)]

        # convolution
        inti_pulse_compressed =
            ifft(fft(inti_zp,1) .* fft(matched_filter_rep,1),1)

        returned_rb = (1:number_of_range_bins) .+ floor(Int,N_pulse/2)
        inti[:,:,b] = inti_pulse_compressed[returned_rb,:]
    end
end
function MTI_3d!(inti,filter,out)
    n_rb, n_pri, n_beams  = size(inti)
    for b in 1:n_beams
        @threads for i = 1:n_rb
            for j =1:n_pri-length(filter)+1
                s = 0.0
                for  (k,f) in enumerate(filter)
                    s += f*inti[i,j+k-1,b]
                end
                out[i,j,b]= s
            end
        end
    end
end
function doppler_filtering!(inti  ::Array{<:Complex,3},
    window ::Array{<:Real,1}
    )
    Nrb = size(inti,1)
    n_beams = size(inti,3)
    @threads for b in 1:n_beams
        weighted_inti = inti[:,:,b] .*transpose(repeat(window, outer = (1,Nrb)))
        inti[:,:,b] = fft(weighted_inti,2)
```

```
    end
end
function noise_gain(window_doppler_filter ::Array{<:Real,1},
    MTI_filter            ::Array{<:Real,1}
    ) ::Array{<:Real,1}

    Nfft = length(window_doppler_filter)
    M = length(MTI_filter)

    noise_gain = fill(0.0, Nfft)
    for k = 0:Nfft-1
        c = exp(-2im*pi*k/Nfft)
        v = fill(0.0im,Nfft+M-1)
        for n = 0:Nfft+M-2
            for i = 0:M-1
                if ((n-i) >= 0) & ((n-i) < Nfft)
                    v[n+1] += MTI_filter[i+1] *
                        window_doppler_filter[n-i+1] * c^(n-i)
                end
            end
        end
        noise_gain[k+1] = sum(abs.(v).^2)
    end
    return noise_gain
end
function equalizer!(inti  :: Array{<:Complex,3},
    window_doppler_filter :: Array{<:Real,1},
    MTI_filter            :: Array{<:Real,1}
    )

    number_of_range_bins = size(inti,1)
    n_beams = size(inti,3)

    noise_gain_per_doppler_channel = noise_gain(window_doppler_filter, MTI_filter)
    equalizer_matrix = transpose(repeat(1 ./ sqrt.(noise_gain_per_doppler_channel),
    @threads for b in 1:n_beams
        inti[:,:,b] = inti[:,:,b] .* equalizer_matrix
    end
end
function create_weights(theta::Array{<:Real},n_channels, lambda,c_type =ComplexF64,
    w = zeros(c_type,n_channels,length(theta))
    for (i,t) in enumerate(theta)
        for k in 1:n_channels
            w[k,i] = exp(1.0im*(k-1)*2*pi*d*sin(t)/lambda)
        end
    end
```

```
    return w
end
function bench(n_rb=512, n_pri=512, n_channels = 128, n_beams = 8,c_type = ComplexF
    inti_header = IntiHeader()

    inti = rand(c_type,n_rb,n_pri,n_channels)
    theta = [(i)*0.01 for i in 0:n_beams-1] #some arbitrary directions
    w = create_weights( theta, n_channels,  inti_header.carrier_wavelength__m,c_typ
    inti2 = zeros(c_type,n_rb,n_pri,n_beams)
    inti3 = zeros(c_type,n_rb,n_pri,n_beams)
    inti4 = zeros(c_type,n_rb,n_pri-1,n_beams)

    # dbf_t_beams2!(inti,inti2,w)
    # Check that dbf is correct)
    dbf_t_beams!(inti,inti2,w)
    dbf_t_beams2!(inti,inti3,w)
    print(inti2 == inti3)

    b1 = @benchmark dbf_t_beams!($inti,$inti2,$w)
    display(b1)
    b2= @benchmark dbf_t_beams2!($inti,$inti2,$w)
    display(b2)
    b3= @benchmark MTI_3d!($inti2,$[-0.7,0.7],$inti4)
    display(b3)

end
function run_multi_beam(n_rb=256, n_pri=256, n_channels = 128, n_beams = 8, iterati
    # Requires
    dims=(n_rb,n_pri,n_channels,n_beams)
    inti_header = IntiHeader()
    runtimest = zeros(UInt64,iterations)
    inti = rand(c_type,n_rb,n_pri,n_channels)
    theta = [(i)*0.01 for i in 0:n_beams-1] #some arbitrary directions
    w = create_weights( theta, n_channels,  inti_header.carrier_wavelength__m,c_typ
    window_dopp = hanning_window(n_pri - 1)
    inti2 = rand(c_type,n_rb,n_pri,n_beams)
    inti3 = rand(c_type,n_rb,n_pri-1,n_beams)
    GC.gc();GC.gc()
    for i in 1:iterations
        #inti = rand(ComplexF64,n_rb,n_pri,n_channels)
        t = time_ns()
        GC.gc(false)
        GC.enable(false) # turn off gc
        dbf_t_beams!(inti,inti2,w)
        pulse_compression!(inti2, inti_header.pulse, inti_header.window_pulse_compr
        #print(size(dcube_post_PC ))
```

```
        MTI_3d!(inti2, inti_header.MTI_filter,inti3)
        #print(size(dcube_post_MTI))
        doppler_filtering!(inti3, window_dopp)
        equalizer!(inti3, window_dopp, inti_header.MTI_filter)
        GC.enable(true)
        runtimest[i]= time_ns()-t
    end
    return round.(1e-6*(runtimest),digits = 3), dims
end
function run_multi_beam_no_gc(iterations = 500,n_rb=256, n_pri=256, n_channels = 12
    # Requires
    dims=(n_rb,n_pri,n_channels,n_beams)
    inti_header = IntiHeader()
    runtimest = zeros(UInt64,iterations)
    inti = rand(c_type,n_rb,n_pri,n_channels)
    theta = [(i)*0.01 for i in 0:n_beams-1] #some arbitrary directions
    w = create_weights( theta, n_channels,  inti_header.carrier_wavelength__m,c_typ
    print(size(w))
    window_dopp = hanning_window(n_pri - 1)
    inti2 = rand(c_type,n_rb,n_pri,n_beams)
    inti3 = rand(c_type,n_rb,n_pri-1,n_beams)
    GC.gc();GC.gc()
    for i in 1:iterations
        #inti = rand(ComplexF64,n_rb,n_pri,n_channels)
        t = time_ns()
        dbf_t_beams!(inti,inti2,w)
        pulse_compression!(inti2, inti_header.pulse, inti_header.window_pulse_compr
        #print(size(dcube_post_PC ))
        MTI_3d!(inti2, inti_header.MTI_filter,inti3)
        #print(size(dcube_post_MTI))
        doppler_filtering!(inti3, window_dopp)
        equalizer!(inti3, window_dopp, inti_header.MTI_filter)
        runtimest[i]= time_ns()-t
    end
    return round.(1e-6*(runtimest),digits = 3), dims
end
function eval_aesa_block!(inti::Array{<:Complex,3},Y::Array{<:Real,3},w::Array{<:Co

    n_rb,n_pri,n_channels = size(inti)
    n_beams =  size(Y,3)
    #runtimest = zeros(UInt64,iterations)
    #println("I got inside")
    #inti = X_real + X_imag*1.0im
    #theta = [(i)*0.01 for i in 0:n_beams-1] #some arbitrary directions
    #w = ones(c_type,n_channels,n_beams)
    window_dopp = hanning_window(n_pri - 1)
```

```
    inti2 = similar(inti,n_rb,n_pri,n_beams)
    inti3 = similar(inti,n_rb,n_pri-1,n_beams)
    #GC.gc(false)
    dbf_t_beams!(inti,inti2,w)
    #GC.gc(false)
    pulse_compression!(inti2, inti_header.pulse, inti_header.window_pulse_compressi
    #print(size(dcube_post_PC ))
    MTI_3d!(inti2, inti_header.MTI_filter,inti3)
    #print(size(dcube_post_MTI))
    doppler_filtering!(inti3, window_dopp)
    equalizer!(inti3, window_dopp, inti_header.MTI_filter)

    Y[:,:,:] = complex2dB.(inti3)
    return 0
end

function test_block(n_rb=128, n_pri=256, n_channels = 16, n_beams = 8, iterations =
    # Requires
    dims=(n_rb,n_pri,n_channels,n_beams)
    inti_header = IntiHeader()
    in = rand(Complex{f_type},n_rb,n_pri,n_channels)
    #n_imag = rand(f_type,n_rb,n_pri,n_channels)
    out = zeros(f_type,n_rb,n_pri-1,n_beams)
    w = ones(Complex{f_type},n_channels, n_beams)
    println("Testing AEAS for dimesions $(dims)")
    @time eval_aesa_block!(in,out,w,inti_header)
    @time eval_aesa_block!(in,out,w,inti_header)
    @time eval_aesa_block!(in,out,w,inti_header)
end

function run_eval_block(n_rb=128, n_pri=256,
n_channels = 16, n_beams = 8, iterations = 500,f_type = Float32)
    # Requires
    dims=(n_rb,n_pri,n_channels,n_beams)
    inti_header = IntiHeader()
    runtimest = zeros(UInt64,iterations)
    inti = rand(Complex{f_type},n_rb,n_pri,n_channels)
    #theta = [(i)*0.01 for i in 0:n_beams-1] #some arbitrary directions
    #w = create_weights( theta, n_channels,  inti_header.carrier_wavelength__m,c_
    w = ones(Complex{f_type},n_channels, n_beams)
    #window_dopp = hanning_window(n_pri - 1)
    #inti2 = rand(c_type,n_rb,n_pri,n_beams)
    out = rand(f_type,n_rb,n_pri-1,n_beams)
    GC.gc();GC.gc()
    for i in 1:iterations
        t = time_ns()
```

```
        GC.gc(false)
        GC.enable(false)
        eval_aesa_block!(inti,out,w,inti_header)
        GC.enable(true)
        runtimest[i]= time_ns()-t
    end
    return round.(1e-6*(runtimest),digits = 3), dims
end
function save_rt(runtimes_c)
    # save runtime data from c in Julia format
    filename = "results/embedded_rt.JLD2"
    @save filename runtimes_c
end



FFTW.set_num_threads(1)
BLAS.set_num_threads(1)
#bench()
#test_block()
rt,dims =run_multi_beam()
@save "results/aesa_gc_threads_$(nthreads()).JLD2" rt dims

rt,dims =run_multi_beam_no_gc()
@save "results/aesa_no_gc_threads_$(nthreads()).JLD2" rt dims

#rt, dims = run_eval_block()
#test_block()

#@save "results/aesa_gc_serial.JLD2" rt dims
#@save "results/aesa_gc_threads_$(nthreads()).JLD2" rt dims
#@save "results/aesa_gc_serial_$(nthreads()).JLD2" rt dims

println("Runtimes [ms] mean : $(mean(rt[10:end])),  min :"*
        "$(minimum(rt[10:end])),  max: $(maximum(rt[10:end])),"*
        "  std=$(round(std(rt[10:end]),digits = 3)),  var=$(round(var(rt[10:end]),d
#println("Done!")
```

A.3.2 Embedded AESA

Source Code A.2: aesaf.c

```c
#include <julia.h>
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>
#include <time.h>
JULIA_DEFINE_FAST_TLS() // only define this once, in an executable
//(not in a shared library) if you want fast code.

// compile with
// gcc -o exe_name -fPIC -I/home/root/Desktop/julia-1.5.3/include/julia
//-L/home/anton/Desktop/julia-1.5.3/lib
//-Wl,-rpath,/home/root/Desktop/julia-1.5.3/lib file_name.c -ljulia


jl_value_t *checked_eval_string(const char* code)
{
    jl_value_t *result = jl_eval_string(code);
    if (jl_exception_occurred()) {
        // none of these allocate, so a gc-root (JL_GC_PUSH) is not necessary
        jl_call2(jl_get_function(jl_base_module, "showerror"),
                 jl_stderr_obj(),
                 jl_exception_occurred());
        jl_printf(jl_stderr_stream(), "\n");
        jl_atexit_hook(1);
        exit(1);
    }
    assert(result && "Missing return value but no exception occurred!");
    return result;
}
int main(int argc, char *argv[])
{
    /* required: setup the Julia context */
    jl_init();


    checked_eval_string("import(Pkg)");
    checked_eval_string("Pkg.activate(\"MPDBasic\")");
    checked_eval_string("using MPDBasic");
    jl_function_t *println = jl_get_function(jl_base_module, "println");
    if (jl_exception_occurred())
        printf("Self made function does not work %s \n",
        jl_typeof_str(jl_exception_occurred()));
```

```
// Hash to store pointers in global scope to keep data safe from GC
jl_value_t* refs = checked_eval_string("refs = IdDict()");
jl_function_t* setindex = jl_get_function(jl_base_module, "setindex!");
jl_datatype_t* reft = (jl_datatype_t*)checked_eval_string("Base.RefValue{Any}")


// Create 3D array of float32 type
//*/
int dim1 = 128;//n_rbins
int dim2 = 256;//n_doppler
int dim3 = 16;//n_channels
int dim4 = 8;//n_beams
checked_eval_string("n_rbins, n_doppler, n_channels ,
    n_beams= (128,256,16,8)");
/*/
int dim1 = 4;//n_rbins
int dim2 = 4;//n_doppler
int dim3 = 4;//n_channels
int dim4 = 2;//n_beams
checked_eval_string("n_rbins, n_doppler, n_channels, n_beams = (4,4,4,2)");
//*/

int iterations = 500;
double runtimes [iterations];
long t;


// Allocate memory for real and imaginary part of a complex matrix

float complex *input =(float complex*)malloc(dim1*dim2*dim3*
                        sizeof(float complex));


float *out = (float*)malloc(dim1*(dim2-1)*dim4*sizeof(float));

//float *out_imag =(float*)malloc(dim1*dim2*sizeof(float));

// Fill memory with random variables

for(size_t i=0; i<dim1; i++){
    for(size_t j=0; j<dim2; j++){
        for(size_t k=0; k<dim3; k++){
            input[i + dim1*j + dim1*dim2*k] = ((float)rand()/RAND_MAX*2.0-1.0)
            +((float)rand()/RAND_MAX*2.0-1.0)*I;
        }
```

```
        //out[i+dim1*j] = 0+0*I;
        }
    }


     // dimensions of in input data
    jl_value_t* dimstup = checked_eval_string("(n_rbins, n_doppler, n_channels)");
    {
    JL_GC_PUSH1(&dimstup);
    jl_value_t* rdimstup = jl_new_struct(reft, dimstup);
    jl_call3(setindex, refs, rdimstup, rdimstup);
    JL_GC_POP();
    }
    // dimensions of output data
    jl_value_t* outdimstup = checked_eval_string("(n_rbins, n_doppler-1,n_beams)");
    {
    JL_GC_PUSH1(&outdimstup);
    jl_value_t* routdimstup = jl_new_struct(reft, outdimstup);
    jl_call3(setindex, refs, routdimstup, routdimstup);
    JL_GC_POP();
    }

    jl_value_t* complex_type_f32 = checked_eval_string("Complex{Float32}");
    {
    JL_GC_PUSH1(&complex_type_f32);
    jl_value_t* rcomplex_type_f32 = jl_new_struct(reft, complex_type_f32);
    jl_call3(setindex, refs, rcomplex_type_f32, rcomplex_type_f32);
    JL_GC_POP();
    }
    // array types of input data

    jl_value_t* arrtype = jl_apply_array_type((jl_value_t*)complex_type_f32,3);
    {
    JL_GC_PUSH1(&arrtype);
    jl_value_t* rarrtype = jl_new_struct(reft, arrtype);
    jl_call3(setindex, refs, rarrtype, rarrtype);
    JL_GC_POP();
    }

    // array types of output data

    jl_value_t* outarrtype = jl_apply_array_type((jl_value_t*)jl_float32_type,3);
    {
    JL_GC_PUSH1(&outarrtype);
    jl_value_t* routarrtype = jl_new_struct(reft, outarrtype);
    jl_call3(setindex, refs, routarrtype, routarrtype);
```

```
JL_GC_POP();
}

// Assign input memory to julia arrays

jl_array_t* X = jl_ptr_to_array(arrtype,input,dimstup,0);
{
JL_GC_PUSH1(&X);
jl_value_t* rX = jl_new_struct(reft, X);
jl_call3(setindex, refs, rX, rX);
JL_GC_POP();
}

// Assign output memory to julia arrays
 checked_eval_string("println(\"Something\")");
jl_array_t* Y = jl_ptr_to_array(outarrtype,out,outdimstup,0);
{
JL_GC_PUSH1(&Y);
jl_value_t* rY = jl_new_struct(reft, Y);
jl_call3(setindex, refs, rY, rY);
JL_GC_POP();
}

// Make sure the data is safe
jl_gc_collect(1);
jl_gc_collect(1);
jl_gc_collect(1);
jl_gc_collect(1);


jl_value_t **args;
JL_GC_PUSHARGS(args, 4);
args[0] = (jl_value_t*)X;
args[1] = (jl_value_t*)Y;
jl_value_t* w =
    checked_eval_string("ones(Complex{Float32},n_channels,n_beams)");
args[2] = w;
jl_value_t* inti_header = checked_eval_string("inti_header = IntiHeader()");
args[3] = inti_header;

checked_eval_string("include(\"kladd/aesa.jl\")");
jl_function_t *eval_aesa_block =
    jl_get_function(jl_main_module, "eval_aesa_block!");


for(size_t i=0; i<iterations; i++){
```

```
        t = clock();
        jl_gc_collect(0);
        jl_gc_enable(0);
        jl_call(eval_aesa_block,args,4);
        //if (jl_exception_occurred())
        //    printf("ERROR with julia function %s \n",
        //    jl_typeof_str(jl_exception_occurred()));
        runtimes[i] = (double)1000*(clock()-t)/CLOCKS_PER_SEC;
        //printf("%6.6f ", runtimes[i]);
        jl_gc_enable(1);
    }


    //printf("Average calculation time %6.6f \n ",
    //(float)(clock()-t)/(iterations*CLOCKS_PER_SEC));



    // save runtimes to a julia array and save with julia
    jl_value_t* array_type1d =
        jl_apply_array_type((jl_value_t*)jl_float64_type, 1);
    jl_array_t* jruntimes =
        jl_ptr_to_array_1d(array_type1d, runtimes, iterations,0);

    jl_function_t *save_func = jl_get_function(jl_main_module, "save_rt");
    jl_call1(println,(jl_value_t*)jruntimes);
    jl_call1(save_func,(jl_value_t*)jruntimes);


    //float *p = (float*)jl_array_data(Y);
    //int ndims = jl_array_ndims(Y);
    //printf("\n%d \n",ndims);
    //jl_aara_t *func = jl_get_function(jl_base_module, "size");

    //jl_call1(println, (jl_value_t*)jl_out_real);

    checked_eval_string("println(\"Julia will now terminate\")");
    jl_atexit_hook(0);
    return 0;
}
```

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY