Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Jacob August Rangnes

Hardware Acceleration of Real-Time Angle of Arrival Positioning

Master's thesis in Electronic Systems Design and Innovation
Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Karl Emil Sandvik Bohne

June 2023

NTNU
Norwegian University of
Science and Technology

EmLogic

Jacob August Rangnes

Hardware Acceleration of Real-Time Angle of Arrival Positioning

Master’s thesis in Electronic Systems Design and Innovation
Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Karl Emil Sandvik Bohne
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Project Description

The thesis deals with hardware acceleration of parts of Angle of Arrival calculations, preferably using the Multiple Signal Classification (MUSIC) algorithm and Field Programmable Gate Array (FPGA). The hardware accelerator shall be designed for use in an already existing system, where a battery driven “tag” communicates wirelessly with a “locator” using Bluetooth Low Energy. The hardware accelerator can be designed specifically for this system, and optimizations of the MUSIC algorithm can be explored. The accelerator shall be evaluated in terms of speed of operation, precision, and power consumption by being compared to a pure software-based implementation.

Abstract

Real-Time Locating Systems (RTLS) have become vital across industries as they allow for effective resource management and optimized logistics. In 2019, Bluetooth enhanced the Bluetooth Low Energy (BLE) technology, allowing for centimeter-level precision for Indoor Positioning Systems (IPS), a subset of RTLS. Allowing for Angle of Arrival (AoA) and Angle of Departure technology (AoD), BLE offers high precision with low complexity and power usage. In this context, a system was developed as a part of *TFE4580 - Specialization Project* to demonstrate the improvements. This thesis aims to further improve this system by implementing a hardware accelerator for the AoA computations.

The Multiple Signal Classification (MUSIC) algorithm is a popular algorithm, and it is widely used for estimating the AoA in RTLS. In this thesis, the work from multiple versions of the MUSIC algorithm is combined, and further optimized for a specific antenna structure and for use in BLE applications. During the optimizations, new methods for deriving a real-valued MUSIC algorithm is explored for non-uniform antenna arrays.

A real-valued transformation of the MUSIC algorithm is explored, allowing the implemented hardware accelerator to achieve greater level of parallelism. The implementation of the algorithm is divided into the work of two theses, where this thesis mainly focuses on the final step of the MUSIC algorithm, where a search is performed, and the goal is to find the AoA. For the search, two options are implemented and compared to each other, and to a series of high-level models, written in Python and C. A multiple-search approach is derived, reducing the complexity for the search, without reducing the search precision significantly.

The power- and time consumption is measured for all implemented versions, allowing us to compare the hardware accelerated search to the high-level models. Compared to Python, which is the language used for implementing the MUSIC algorithm in the previously implemented system, significantly time reduction is observed, reducing the execution time from 154 ms to 5.34 µs. The hardware accelerated search is also significantly more energy efficient when compared to the high-level models. Energy efficiency is essential when implementing low-power systems, and with the obtained results, the hardware accelerated search would further improve the already existing system in terms of speed and energy efficiency while maintaining approximately the same level of precision. Some errors are introduced when specific AoAs are present, and the reason for this is due to the values used in the search being limited in terms of decimal precision.

The implemented search is a part of a larger design, where all steps of the MUSIC algorithm are implemented on an FPGA. The implementation of the remaining parts of the algorithm can be found in the thesis written by Tommy A. Opstad [1]. The two designs are merged together to confirm that the two designs are compliant with each other, and an estimation of the total performance is given.

Sammendrag

Sanntids-lokasjonssystemer har blitt viktig i en rekke bransjer da de muliggjør for effektiv utstyrskontroll og optimalisert logistikk. I 2019 forbedret Bluetooth sin BLE teknologi, slik at det ble mulig med presisjon på centimeternivå for innendørs posisjoneringssystemer (IPS), en undergruppe av RTLS. Ved å støtte AoA og AoD teknologi tilbyr BLE høy presisjon med lav kompleksitet og strømforbruk. I denne sammenhengen ble et system utviklet som en del av *TFE4580 - Fordypningsprosjekt* for å demonstrere den nye teknologien. Denne oppgaven har som mål å videre forbedre dette systemet ved å akselerere AoA-beregninger.

MUSIC algoritmen er en populær algoritme og brukes ofte for å estimere AoA i RTLS. I denne oppgaven blir arbeidet fra flere versjoner av MUSIC-algoritmen kombinert og ytterligere optimalisert for en spesifikk antennekonfigurasjon og for bruk i BLE-applikasjoner. Under optimaliseringen utforskes nye metoder for å utlede en reell-MUSIC algoritme for ikke-uniforme antennestrukturer.

En transformasjon av MUSIC-algoritmen, hvor målet er å gjøre alle verdier reelle, blir utforsket, slik at den implementerte akseleratoren kan oppnå høyere grad av parallellitet. Implementeringen av algoritmen er delt inn i to masteroppgaver, der denne oppgaven hovedsakelig setter søkelys på det siste trinnet i MUSIC-algoritmen, der et søk utføres for å finne AoA. For søket er to alternativer implementert og sammenlignet med hverandre og med en serie av høy-nivå modeller, skrevet i Python og C. En flersøk-tilnærming er utviklet, som reduserer kompleksiteten for søkningen uten å redusere presisjonen vesentlig.

Effekt- og tidsforbruket er målt for alle implementerte versjoner, noe som gjør det mulig å sammenligne den akselererte algoritmen med høy-nivå modellene. Sammenlignet med Python, som er språket som ble brukt til å implementere MUSIC-algoritmen i det tidligere implementerte systemet, ble det observert betydelig tidsreduksjon, med en kjøretid som ble redusert fra 154 ms til 5,34 µs. Det akselererte søket er også betydelig mer energieffektivt sammenlignet med høy-nivå modellene. Energieffektivitet er viktig ved implementering av lavenergisystemer, og med de oppnådde resultatene vil det akselererte søket ytterligere forbedre det allerede eksisterende systemet når det gjelder hastighet og energieffektivitet, samtidig som omtrentlig samme presisjon blir opprettholdt.

Søket som er implementert er en del av et større design, hvor alle stegene i MUSIC algoritmen er implementert på en FPGA. De gjenværende stegene av algoritmen er implementert av Tommy A. Opstad, og mer informasjon om de stegene kan bli funnet i hans masteroppgave [1]. Resultatene fra de to oppgavene er satt sammen for å sørge for at designene er kompatible med hverandre, og et estimat på den totale ytelsen av den totale akselerasjonen er gitt.

Preface

This thesis is the final requirement to be awarded the title Master of Science in Electronic Systems Design and Innovation from the Norwegian University of Science and Technology. It is a continuation of the project thesis carried out in the preceding semester. Both projects have been performed on behalf of EmLogic, a rapidly growing design center for embedded systems. It has been a great pleasure to work with you! A special thanks to my supervisor, Karl Emil Sandvik Bohne for your continuous support throughout this semester. I would also like to thank Espen Flo Eriksen for helping me understand some of the challenging mathematical problems I have encountered while working with this thesis. Both projects have also been carried out under superb academic supervision from Prof. Per Gunnar Kjeldsberg at the Norwegian University of Science and Technology. I am grateful for your feedback and for the ideas of topics to discuss given through reviews and meetings.

During both projects, I have worked closely with another student, Tommy A. Opstad. His hard work and dedication has played an important role in both projects' success. It is clear that you enjoy working with digital- and embedded systems, and I wish you all the best in the future.

Lastly, I would also like to use the opportunity to thank my girlfriend, Nora, for your continuous support and encouragement throughout these five years. These five years in Trondheim went by really fast, and we have made memories that will last forever. I am forever grateful for all the reports you have helped me proofread and the exams you have helped me prepare for. I acknowledge that understanding the things I am working with can be challenging, but don't worry; we have plenty of time for that.

Jacob A. Rangnes

Jacob A. Rangnes
Trondheim, June 2023

Contents

Appendices

List of Tables

List of Figures

List of abbreviations

AWGN Additive White Gaussian Noise

AoA Angle of Arrival

AXI Advanced eXtensible Interface

BLE Bluetooth Low Energy

BRAM Block RAM

BG BRAM Group

CMC Covariance Matrix Calculation

CTE Constant Tone Extension

DDR3 Double Data Rate 3

DSP Digital Signal Processing

DUT Device Under Test

EVD Eigenvalue Decomposition

FPGA Field Programmable Gate Array

FB Forward/Backward

GFSK Gaussian Frequency Shift Keying

GPS Global Positioning Systems

FSK Frequency Shift Keying

ISM Industrial, Scientific and Medical

IQ In-Phase and Quadrature

IPS Indoor Positioning Systems

LL Link Layer

MUSIC Multiple Signal Classification

MAC Multiply-Accumulate

PDU Protocol Data Unit

PS Processing System

RAM Random Access Memory

RF Radio Frequency

RV Real-Valued

RVT Real-Valued Transformation

RTLS Real-Time Locating Systems

SoC System on Chip

SPS Spectral Peak Search

TB Testbench

ULA Uniform Linear Array

URA Uniform Rectangular Array

VECMUL Vector Multiplication

Chapter 1

Introduction

1.1 Motivation

The demand for Real-Time Locating Systems (RTLS) has grown significantly in recent years due to their potential in enhancing security and improving operational efficiency. RTLS are defined by the ISO/IEC 24730-1:2014 standard [2], and are included in the 9th sustainability goal, *Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation* [2, 3]. The technology is intended for tracking and monitoring of assets, people, and vehicles in real-time, providing essential information for critical decision-making processes. As an example, using RTLS in emergency response allow the operators to locate the nearest personnel and equipment, reducing the response time for critical situations.

Indoor Positioning Systems (IPS) are a subset of RTLS, and are used for locating objects in closed structures where other RTLS technologies, such as Global Positioning Systems (GPS), generally have poor performance [4]. Figure 1.1 illustrates how an IPS are typically implemented for tracking assets in warehouses. By using wireless technology, we can place transmitters on the assets to be tracked, communicating with receivers placed at strategic locations within the closed structure.

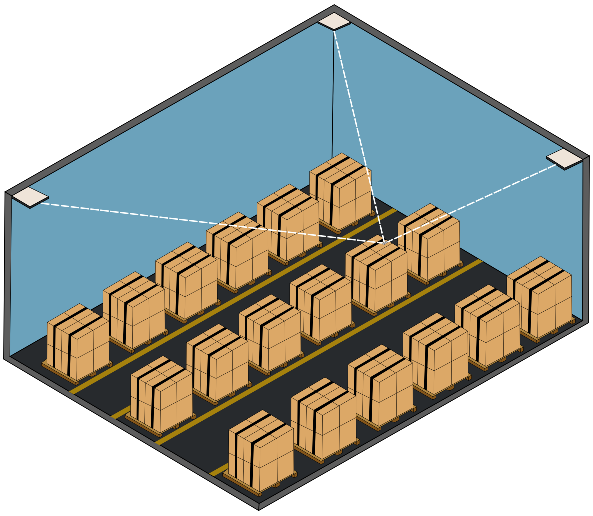


Figure 1.1: Example of IPS used in a warehouse.

In 2019, improvements were introduced for the Bluetooth technology to enhance the performance of IPS using the Bluetooth Low Energy (BLE) technology, reportedly allowing for centimeter-level precision [5]. Bluetooth is a widely adopted technology with approximately 400 million devices shipped yearly to be used in RTLS [5].

However, the implementation of RTLS poses several challenges, including the need for high computational power and low latency. These challenges can be addressed by using hardware accelerators, which can speed up the processing time of RTLS algorithms and reduce power consumption compared to pure software-based solutions. In this context, this thesis proposes the implementation of a hardware accelerator for the Multiple Signal Classification (MUSIC) algorithm, a widely used algorithm in RTLS.

1.2 Main Objectives

The main objectives for the thesis are summarized in the list below.

1. What optimizations of the MUSIC algorithm can be done for use in BLE applications and specific antenna structures?
2. How does the optimized MUSIC algorithm perform when running on an FPGA compared to a pure software-based implementation in terms of speed, precision and power usage?

1.3 Methodology

The strategy for this thesis is summarized in Figure 1.2.

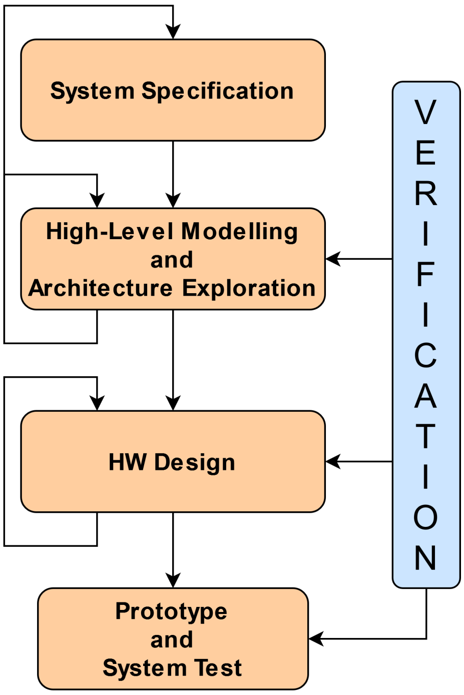

System Specification
High-Level Modelling and Architecture Exploration
HW Design
Prototype and System Test
VERIFICATION

Figure 1.2: Methodology for the thesis.

The initial phase of the thesis was dedicated to literature study with a goal of understanding and experimenting with the MUSIC algorithm. Python [6] is used for making high-level models and has allowed for rapid prototyping and verification of ideas. The implementation of the hardware accelerator is written in VHDL. Simulations, synthesis and implementation are performed using the Xilinx Vivado tool. For verification of the implemented hardware accelerator, Python is used for generating stimuli and expected results. Assertions are written to ensure that the correct behavior is achieved for all test cases.

1.4 Main contributions

- Modification of the real-valued MUSIC algorithm, provided in [7], to fit non-Uniform Rectangular Array (URA)- or Uniform Linear Array (ULA) antenna structures.
- Observed symmetrical properties of the real-valued steering vectors, reducing the needed memory size by 50%.
- Proposed multiple hardware architectures for the real-valued MUSIC search function.
- Implemented two solutions for the real-valued MUSIC search function, both offering significant speedup compared to a software-based implementation.
- Observed and discussed how the quantization of numbers influence the precision of the hardware accelerated MUSIC search.
- Cooperated with Tommy A. Opstad to implement a complete real-valued MUSIC algorithm on the Xilinx Z-7020 System on Chip.
- Presented theoretical improvements for speedup and energy consumption, and introduced a discussion on how the operating frequency can be adjusted to fit the desired usage for the hardware accelerator.
- Implemented parts of the MUSIC algorithm in Python, C and VHDL in order to compare the performance of the hardware accelerator.

1.5 Report Outline

Chapter 2 Background contains a more detailed description of the system where the hardware accelerator is to be implemented. To understand how this system works, an introduction to Bluetooth Low Energy (BLE) and the technology allowing for BLE to be used in IPS is also provided here.

Chapter 3 Theory provides all relevant theory for the implementation of the hardware accelerator. The main focus for this chapter is to present the MUSIC algorithm. To fully understand the algorithm, relevant matrix theory is also provided, as the algorithm introduces various matrix operations. This chapter first presents the original, complex-valued, MUSIC algorithm before presenting a real-valued approach of the same algorithm.

Chapter 4 Derivation of application-specific MUSIC algorithm presents the derivation of the application-specific vectors and matrices. This chapter contains further modifications of the results of the theory presented in Chapter 3.

Chapter 5 Implementation first presents the structure of the hardware accelerator to be designed. The methods used for solving the given tasks are discussed before multiple possible architectures are presented and discussed.

Chapter 6 Test and Results describes how the implemented accelerator is tested and verified. To evaluate and discuss the performance of the accelerator, the same tests are also performed on a high-level model. These results are also presented in this chapter.

Chapter 7 Discussion provides a discussion of the obtained results. The discussion is focused around the objectives presented in Section 1.2. A recommendation for future improvements are also given in this chapter.

In **Chapter 8 Conclusion** concludes the work done in this thesis.

Chapter 2

Background

This thesis is a continuation of the project carried out in the preceding semester as a part of `TFE4580 - Specialization Project`, where a system was made to demonstrate the improvements of the BLE technology included with the Bluetooth Core Specification v5.1 [5]. The system was implemented by the author of this thesis [8] and Tommy A. Opstad [9]. The following section will briefly present the BLE technology and discuss the relevant parts of the implemented system and the obtained results.

2.1 System overview

The previously implemented system consists of three main components; a tag, a locator, and a computer estimating- and visualizing the location of the tag. An overview of how the components communicates is shown in Figure 2.1.

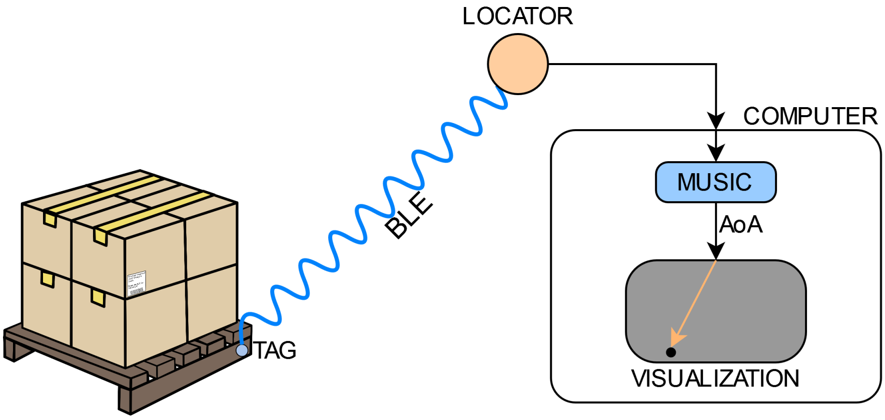


Figure 2.1: An illustration of the previously implemented system.

As indicated in the figure, a tag communicates wirelessly with a locator by using BLE. The locator receives the signal on multiple antennas and further communicates the received values to a computer that is running a Python program for estimating and visualizing the position of the tag [8, 9]. The system allows multiple tags to communicate with one locator board. The individual tags are assigned an ID, and they are using an accelerometer to detect movement. The detected movement is used as a trigger to rapidly transmit multiple BLE packets for a short period of time before returning to sleep [8]. This version of the system is using the MUSIC algorithm for estimating the position. Further details on this algorithm are presented in Section 3.6. Both the tag and locator use the nRF52833 SoC as the chosen processor.

A photograph of the assembled system is presented in Figure 2.2. The height difference between the tag and the locator is known and constant, $h = 130$ cm [8]. This height can be, together with the estimated direction for the received signal, used for calculating the position of the tag in the two-dimensional plane. For this system, the movement of the tag is limited to only move in two dimensions.

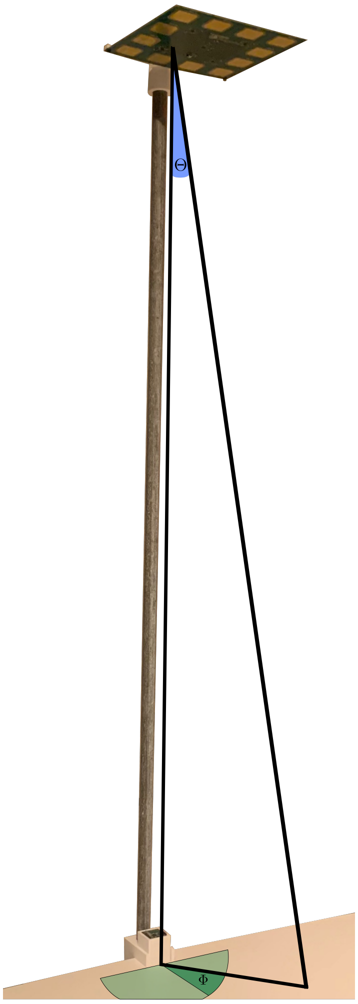


Figure 2.2: A photograph of the assembled system.

2.2 Bluetooth Low Energy

As indicated in Section 2.1, the tag communicates with the locator by using BLE. This section presents the fundamentals of the BLE technology and how it can be used in direction finding applications.

With the Bluetooth Core Specification version 5.1 [10], new features were introduced to the

BLE controller shown in Figure 2.3. The relevant changes were done in the Link Layer, reportedly allowing for centimeter-level precision estimations of the direction of a received- or transmitted signal [5]. The remaining of this section will describe the relevant parts of the BLE stack and explain how they allow for BLE to be used in direction finding applications.

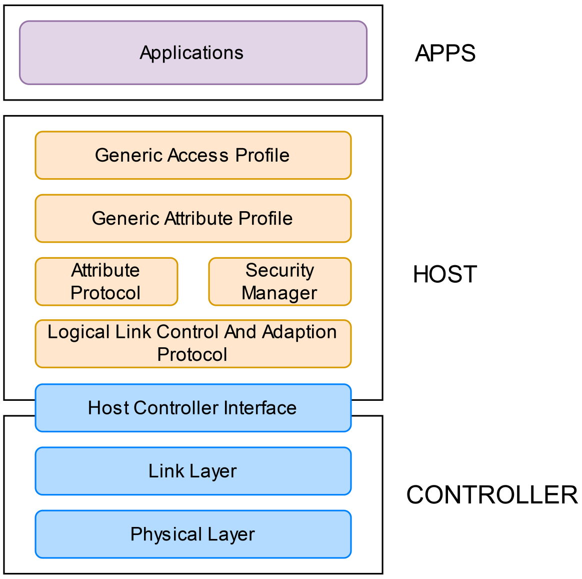


Figure 2.3: The BLE stack.

2.2.1 Physical Layer

The physical layer is the lowest layer in the BLE stack, and is the layer where the connection and communication between devices are made. BLE is located in the 2.4GHz Industrial, Scientific and Medical (ISM) band, divided into 40 1MHz-channels as shown in Figure 2.4 [10, p. 2660]. There are two types of channels, advertising- and data channels. The former is used only for broadcasting data, while the latter can be used for the same operations or communicating with specific devices [10, p. 2690].

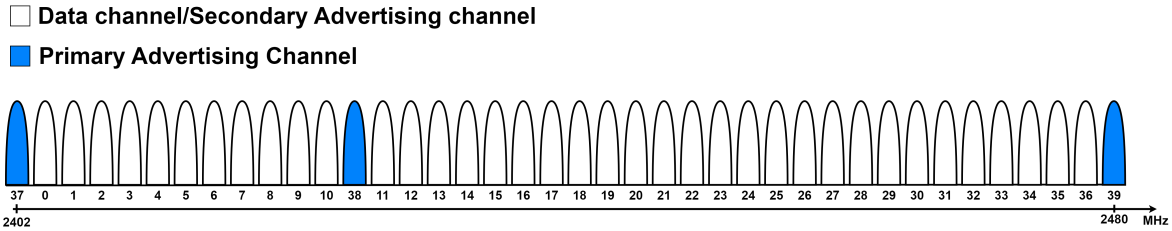


Figure 2.4: Visual representation of the 40 BLE channels.

From Figure 2.4, it can be seen that the frequency for each band is different, resulting in a difference in the wavelength for each of the channels. The relation between the frequency, f, velocity, v, and wavelength, λ, is given by [11, p. 341]

$$\lambda = \frac{v}{f}. \tag{2.1}$$

This means that for the implemented system presented in Section 2.1, we can expect to receive data on all the available channels, resulting in a non-constant wavelength for the received BLE packets. This will later be important for the MUSIC algorithm presented in Section 3.6, and a discussion of the effects is presented in Section 7.5.

For modulation of the data, BLE mainly uses Gaussian Frequency Shift Keying (GFSK) [10, p. 2662]. GFSK is a technique used for modulating wireless data, and it is based on the Frequency Shift Keying (FSK) modulation scheme, where the frequency is shifted [12] to represent binary data. A binary one is represented by a positive frequency deviation, while a binary zero is represented by a negative frequency deviation from the center frequency of the chosen channel, presented in Figure 2.4 [10, p. 2662].

2.2.2 Link Layer

The Link Layer (LL) is the second-lowest layer in the BLE stack presented in Figure 2.3, and it controls the behavior of the BLE controller The LL can be described as a state machine with the available states that are presented in Table 2.1 [10, p. 2682].

Table 2.1: BLE Link Layer states.

Number	State name
1	Standby
2	Advertising
3	Scanning
4	Initiating
5	Connection
6	Synchronization

In state 2, 5, and 6, the controller can transmit packets. The general structure of a BLE packet is presented in Figure 2.5. Depending on the given state for the BLE controller, the transmitted Protocol Data Unit (PDU) is modified and transmitted on either advertising- or data channels [10].

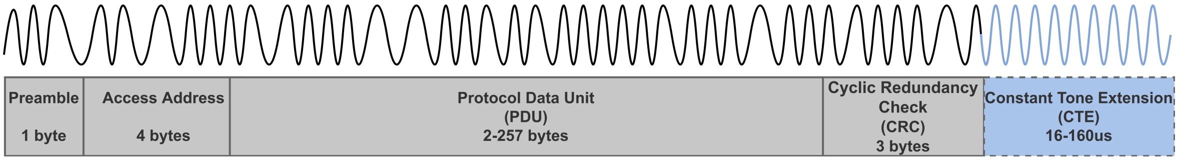
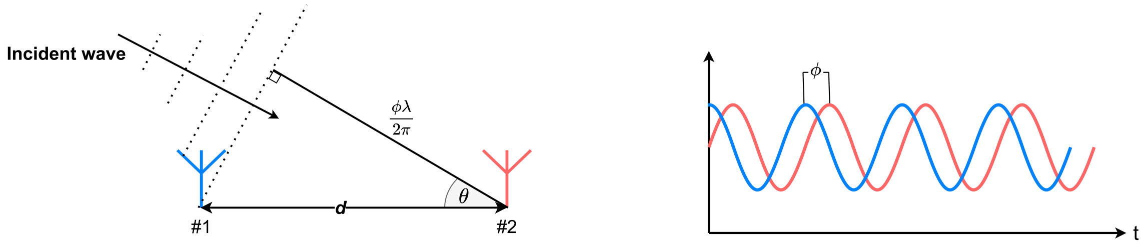


Figure 2.5: BLE Link Layer packet format. The black curves indicate that the transmitted data is modulated using GFSK, while the blue curves are used for indicating a series of modulated binary ones.

The details of the first four sections of the packet, marked with gray, are not presented as they are not relevant for this thesis. However, the Constant Tone Extension (CTE) is highly relevant and will further be presented.

Angle Of Arrival

Before the CTE in the BLE packet is presented, an introduction to the term Angle of Arrival (AoA) is given in the following section. AoA is a term used for a popular technique used in Radio Frequency (RF) systems, allowing us to obtain the direction of the received signal. Multiple properties of the signal can be used for obtaining the AoA, such as time- and phase difference [13]. For BLE, the latter property is used [10, p. 2733]. The fundamental scenario of an incident wave arriving at two antennas spaced with a distance d apart is shown in Figure 2.6(a), and a visualization of how the signal is observed on the two antennas in time domain is shown in Figure 2.6(b).



(a) Incident wave received on two antennas.

(b) Observed signal values for the two antennas.

Figure 2.6: Angle of Arrival between two antenna elements.

From Figure 2.6(a), we observe that a right triangle is formed, and by using trigonometry, we can obtain the AoA [10, p. 282]

$$\theta = \arccos\left(\frac{\phi\lambda}{2\pi d}\right), \tag{2.2}$$

where ϕ is the phase difference between the two observed phase values.

As indicated by this scenario, using two antennas allow us to obtain one angle, resulting in a direction in two dimensions. To obtain a direction in three dimensions (3D), a third element can be added perpendicular to the array in Figure 2.6(a), adding a new AoA in the perpendicular plane to θ in Equation (2.2). By combining these angles, we obtain the 3D AoA. This implies that we only need three antennas for obtaining the AoA in three dimensions. In these cases, the mathematics for estimating the AoA become fairly simple. However, increasing the number of antennas in the array make the estimations of AoA less prone to errors due to noise, as it allows us to observe the AoA between multiple antenna pairs, as in Figure 2.6(a). In the case where a 2D antenna array, containing multiple elements, is used, Equation (2.2) becomes inefficient and inaccurate. Specific algorithms, such as the MUSIC algorithm, are generally more precise in such scenarios. The theory for the MUSIC algorithm is presented in Section 3.6.

CTE

The CTE was added to the BLE packet, presented in Figure 2.5, with Bluetooth Core Specification v.5.1 [10].

The CTE is, as the name indicates, a constant tone which is configurable in length from 16-160 µs. While the data in a BLE packet are modulated using GFSK, the CTE is a series of modulated binary ones [10, p. 2693]. This is illustrated in Figure 2.5. As presented in the above section, this constant tone can be used for obtaining the AoA of the signal, as it allows us to observe the phase difference between antenna elements.

BLE is often used in low-power systems where the cost and power usage should be kept as low as possible. Ideally, the signal values of each antenna in the receiving array should be sampled simultaneously. This is due to that if the antenna signals are sampled sequentially, a time difference will influence the observed phase difference between the two measurements of the antennas. When working with large antenna structures, adding support for fully parallel sampling of the antennas is not feasible. To address this issue, Bluetooth Special Interest Group has proposed a specific sampling routine [10, p. 2733], presented in Figure 2.8, using RF switches as indicated in Figure 2.7.

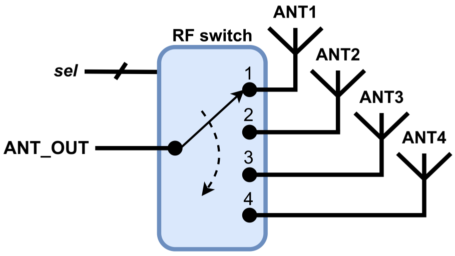


Figure 2.7: Illustration of CTE sampling setup.

The receiving processor switches between the elements in the array with time slots of either 1 µs or 2 µs. The total number of samples per antenna depends on the length of the transmitted CTE and the duration of the time slots for sampling. An overview of how the CTE is transmitted and sampled is shown in Figure 2.8 [10, p. 2733].

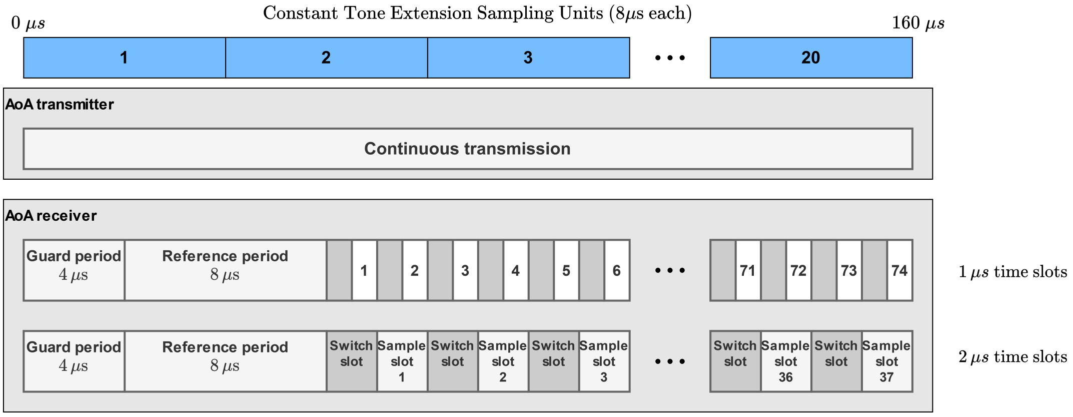


Figure 2.8: Timing diagram for CTE transmission and sampling.

During the `Guard Period`, no sampling of antenna signals is performed, but during the `Reference Period`, one sample is taken every microsecond without switching between the antennas in the array. Eight samples are therefore taken during the reference period for both of the available time slot configurations. The samples taken in the reference period can be used for adjusting the rest of the sampled values. As discussed above, sampling the antenna-elements sequentially introduces a phase difference due to the time difference between the samples. This phase difference is found in the reference period and can be taken into account in the rest of the samples. Figure 2.8 indicates that 1 µs time slots allow for more samples to be taken. However, this often requires better hardware in terms of acquisition time. It is critical that the antenna signal is stable before a sample is taken. Figure 2.9 indicates that the `Sampling Window` of the sampling slots is constant and equal to 0.75 µs for both 1- and 2 µs time slots. An additional microsecond is present to the 2µs sampling slot, allowing the signal to stabilize further before the sampling window is entered.

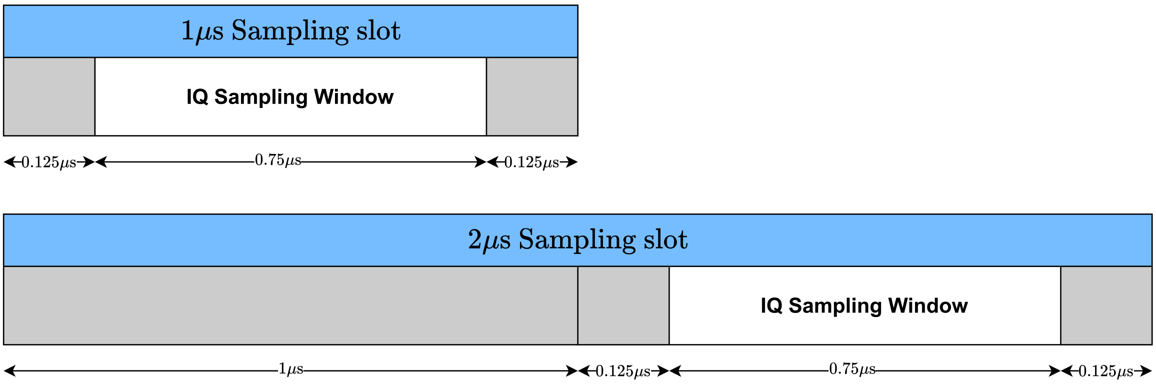


Figure 2.9: Comparison of the sampling window for the two available time slots.

From the above information, a formula for calculating the number of snapshots per antenna, N, can be derived. If D denotes the number of antennas in the array, then

$$N = \text{floor}\left(\frac{\text{CTE length} - 12\text{µs}}{(\text{Switch slot} + \text{Sampling slot}) \cdot D}\right). \tag{2.3}$$

2.3 Results From the Project Thesis

This section is used to present the results from the implementation of the above described system. During the project, different configurations of the BLE controller were discussed and tested [8], while this section will only present the final implementation of the system.

From a series of tests with different variations of CTE transmission time and sampling time slots, the accuracy was tested. The most accurate configuration was found for CTE length of 160 µs and sampling time slots of 2 µs. The tests indicated that the system became more unstable when time slots of 1 µs were used. No major difference in the current consumption of the transmitting component was measured for different CTE lengths. Using the former configuration, centimeter-level precision for the estimated coordinates of the tag was achieved [8], as claimed by Bluetooth SIG [5]. By using Equation (2.3) and the chosen system configurations for the CTE transmission and receiving, we can calculate the expected value of the number of snapshots per antenna, N. The antenna array used for this system has $D = 12$ elements, and is further presented in Chapter 4.

$$N = \text{floor}\left(\frac{160\ \text{µs} - 12\text{µs}}{(2\text{µs} + 2\text{µs}) \cdot 12}\right) = \text{floor}\left(\frac{152}{48}\right) = 3 \tag{2.4}$$

This means that we are able to sample each antenna three times per received CTE.

2.4 PYNQ Z1 Board

The implementation presented in Chapter 5 is designed for the PYNQ Z1 board from Digilent [14]. The board is designed to be used with PYNQ [15], an open-source framework that allows developers to exploit the capabilities of the Xilinx Zynq System on Chip (SoC) [14]. The PYNQ Z1 board offers multiple features, where the main component is the FPGA, ZYNQ XC7Z020 SoC [16]. This section is used for introducing the relevant resources for the SoC used in the implementation in Chapter 5.

2.4.1 RAM36E1

The PYNQ Z1 board offers two main types of memory, Double Data Rate 3 (DDR3) and Block RAM (BRAM). The available BRAM is located within the SoC, and it allows for easy interfacing with custom designs. This section is used for presenting the fundamental properties for the available BRAM resource, RAMB36E1.

The ZYNQ XC7Z020 contains 140 BRAM of size 36 Kb [16], providing great flexibility for the designer to structure the memory in many configurations. The blocks support both dual- and single port operations [17], meaning that one can read from- or write to two addresses within the same BRAM at the same clock cycle, but the BRAMs can also be configured to have only one port for read- and write operations. The width of the data is also configurable, allowing up to 72 bits to be read- or written each clock cycle [16, 17]. The possible configurations are shown in Table 2.2.

Table 2.2: RAMB36E1 layout options [17]. Ki = 1024 values.

Number of values	Data width
32 Ki	1
16 Ki	2
8 Ki	4
4 Ki	9
2 Ki	18
1 Ki	36

2.4.2 DSP48E1

The Digital Signal Processing (DSP) slices available on the ZYNQ XC7Z020 allow us to perform low-power DSP operations, such as multiplication, with high speed and efficiency [18]. The available DSP slices also include other operands, but only the multiplier is used for the design presented in Chapter 5. The ZYNQ XC7Z020 SoC contains 220 DSP slices [16].

2.5 Previous Work

The MUSIC algorithm has, since it was proposed by Schmidt in 1986 [19], been implemented for a variety of applications, providing different methods and optimizations [20, 21, 22, 23].

This thesis reuses, combines, and further optimizes the MUSIC algorithm provided by Keh-Chiarng Huarng and Chien-Chung Yeh in [24], Zhang et al. in [7], Si et al. in [25], and Huang et al. in [26].

Chapter 3

Theory

This chapter presents relevant theory for the implementation presented in Chapter 5.

3.1 Complex Number Arithmetic

Assume that we have two complex numbers, $Z_1 = a + jb$ and $Z_2 = c + jd$, with $a, b, c, d \in \mathbb{R}$. Then [27]

$$\textbf{Addition: } Z_3 = Z_1 + Z_2 = (a + jb) + (c + jd) = (a + c) + j(b + d) \tag{3.1a}$$

$$\textbf{Subtraction: } Z_4 = Z_1 - Z_2 = (a + jb) - (c + jd) = (a - c) + j(b - d) \tag{3.1b}$$

$$\textbf{Multiplication: } Z_5 = A \cdot B = (a + jb)(c + jd) = (ac - db) + j(ad + cd) \tag{3.1c}$$

$$\textbf{Conjugate: } \overline{Z_1} = \overline{a + jb} = a - jb\,. \tag{3.1d}$$

3.2 Representation of Real-Valued Numbers

Representation of real-valued numbers become an important factor when discussing the implementation in Chapter 5, and the supporting theory this discussion is presented in the below subsections. There are two well known methods for representing real-valued numbers, floating- and fixed point numbers. They are both based on the well known binary representation of a real-valued number Equation (3.2).

$$\begin{array}{cccccccccccccc} ... & 32 & 16 & 8 & 4 & 2 & 1 & . & \frac{1}{2} & \frac{1}{4} & \frac{1}{8} & \frac{1}{16} & \frac{1}{32} & ... \\ ... & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & . & 2^{-1} & 2^{-2} & 2^{-3} & 2^{-4} & 2^{-5} & ... \end{array} \tag{3.2}$$

3.2.1 Floating Point Numbers

There are multiple definitions of floating point numbers. While the idea behind the different definitions is very similar, different naming conventions are often used. This thesis will present and use the IEEE 754 standard [28] as the definition. A visualization of how binary data is divided into sign, biased exponent, and trailing significand bits is shown in Figure 3.1.

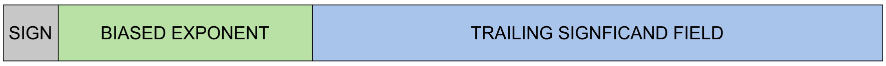


Figure 3.1: IEEE 754 standard for floating point number representation.

Floating point numbers follows the same idea as for scientific notation in base 10. For the well known number

$$(2.99 \cdot 10^8)_{10},$$

we define 8 as the exponent and 2.99 as the significand. Floating point follows the same notation, only in base 2, i.e., binary numbers. The number of bits assigned to the significand determines the precision of the real number, while the exponent determines the range of the number.

Example of IEEE 754 Floating Point Number

For the given bits

0	1	0	0	1	1	0	1	1	0	0	0	1	1	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	1	1	0

,

we observe that the sign bit is set to 0, indicating that the value is positive.

The exponent is

$$10011011_2 = 155_{10},$$

and subtracting the bias of 127 [28, p. 23], we find the final exponent of $155 - 127 = 28$.

The value is therefore found by moving the floating point 28 times to the right, resulting in the final value

$$(+1.00011101001001100000110)_2 \cdot 2^{28} = (10001110100100110000011000000.00)_2 = (2.99 \cdot 10^8)_{10}$$

3.2.2 Fixed Point Numbers

Another common way to represent real-valued numbers is with fixed point numbers. The structure of a fixed point value is shown in Figure 3.2.

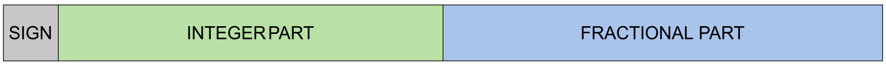


Figure 3.2: Fixed point number representation.

The binary representation follows the standard definition in Equation (3.2).

Example of Fixed Point Number
The 16-bit value

0	1	1	1	1	0	1	1	1	0	0	1	0	0	0	1

,

has the sign bit set to 0, while the integer part is

$$(1111011)_2 = 123_{10},$$

and the fractional part is

$$(0.10010001)_2 = \frac{1}{2} + \frac{1}{16} + \frac{1}{256} \approx 0.567_{10}.$$

This results in the final value being ≈ 123.567.

3.3 Matrix Operation and Theory

In Section 3.6 and Section 3.7, the MUSIC algorithm is presented, introducing various matrix operations. The following subsections present the theory for the introduced operations, together with relevant properties and examples.

3.3.1 Matrix Transpose

The transpose operator is denoted with $(\,\cdot\,)^T$, and the transpose of the matrix $\mathbf{A} = a_{ij}$ is defined as

$$\mathbf{A}^T = a_{ji}, \tag{3.3}$$

meaning that the rows are exchanged for columns and vice versa [29, p. 6].

Equation (3.4) includes an example of matrix transpose of the matrix $\mathbf{A}$.

$$\mathbf{A}^T = \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \\ a_{13} & a_{23} \end{bmatrix}^T = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \tag{3.4}$$

3.3.2 Conjugate operator

The element-wise conjugate operator, denoted with $(\,\cdot\,)^*$, transforms each element to its complex conjugate. That is, if we define $\mathbf{A} = a_{ij}$, then

$$\mathbf{A}^* = \overline{a_{ij}}, \tag{3.5}$$

where $\overline{a_{ij}}$ is the conjugate operation presented in Equation (3.1d).

3.3.3 Hermitian Adjoint

The Hermitian adjoint, denoted with $(\ \cdot\ )^H$, combines the conjugate operator in Equation (3.5) and the transpose rule presented in Equation (3.3). In addition to exchanging columns for rows and vice versa, element-wise conjugate is also applied [29, p. 6]. This means, if all elements in the matrix are real, then the Hermitian adjoint is the same as the transpose. Let $\mathbf{A} = a_{ij}$. Then

$$\mathbf{A}^H = (a_{ij})^H = (\mathbf{A}^*)^T = \left(\mathbf{A}^T\right)^* = \overline{a_{ji}}. \tag{3.6}$$

Equation (3.7) includes an example of the Hermitian adjoint of a complex valued matrix.

$$\begin{bmatrix} 1+j2 & 2+j3 \\ 3+j4 & 4+j5 \\ 5-j6 & 6-j7 \end{bmatrix}^H = \begin{bmatrix} 1-j2 & 3-j4 & 5+j6 \\ 2-j3 & 4-j5 & 6+j6 \end{bmatrix} \tag{3.7}$$

3.3.3.1 Relevant Properties

- **Reverse-order law** [29, p. 6]:

$$(\mathbf{AB})^H = \mathbf{B}^H\mathbf{A}^H. \tag{3.8}$$

- The double Hermitian adjoint of the matrix $\mathbf{A}$ is equal to the original matrix [29, p. 6]:

$$(\mathbf{A}^H)^H = \mathbf{A}. \tag{3.9}$$

3.3.4 Kronecker product

The Kronecker product between two matrices, $\mathbf{A}$ and $\mathbf{B}$, is denoted with $\mathbf{A} \otimes \mathbf{B}$. If $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{B}$ is a $p \times q$ matrix, then $\mathbf{A} \otimes \mathbf{B}$ is the $mp \times nq$ block matrix [29, pp. 474-475]. Each element a_{ij} is multiplied with the matrix $\mathbf{B}$. Consider the two matrices

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}, \tag{3.10}$$

then the Kronecker product is

$$\mathbf{A}\otimes\mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & a_{13}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & a_{23}\mathbf{B} \end{bmatrix} = \begin{bmatrix} a_{11}\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} & a_{12}\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} & a_{13}\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} \\ a_{21}\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} & a_{22}\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} & a_{23}\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} \end{bmatrix}$$
$$= \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} & a_{13}b_{11} & a_{13}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} & a_{13}b_{21} & a_{13}b_{22} \\ a_{11}b_{31} & a_{11}b_{32} & a_{12}b_{31} & a_{12}b_{32} & a_{13}b_{31} & a_{13}b_{32} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} & a_{33}b_{11} & a_{33}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} & a_{33}b_{21} & a_{33}b_{22} \\ a_{21}b_{31} & a_{21}b_{32} & a_{22}b_{31} & a_{22}b_{32} & a_{33}b_{31} & a_{33}b_{32} \end{bmatrix}. \tag{3.11}$$

Relevant properties of the Kronecker Product

1. **The mixed-product property**: If matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and $\mathbf{D}$ are matrices with dimensions such that the matrix-products $\mathbf{AC}$ and $\mathbf{BD}$ can be formed, then $\mathbf{AC}\otimes\mathbf{BD} = (\mathbf{A}\otimes\mathbf{B})(\mathbf{C}\otimes\mathbf{D})$

3.3.5 Identity and Exchange Matrix

This subsection defines two well known matrices. The first is the identity matrix, which has all ones along its northwest-to-southeast diagonal and all zeroes elsewhere [30, p. 50], that is

$$\mathbf{I}_n = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}. \tag{3.12}$$

The second matrix is the exchange matrix, and it has all ones on the opposite diagonal when compared to $\mathbf{I}$ [30, p. 193], and can also be referred to as a reversal matrix or backwards identity matrix. The matrix is defined in Equation (3.13).

$$\mathbf{J}_n = \begin{bmatrix} 0 & \dots & 0 & 0 & 1 \\ 0 & \dots & 0 & 1 & 0 \\ 0 & \dots & 1 & 0 & 0 \\ \vdots & ⋰ & \vdots & \vdots & \vdots \\ 1 & \dots & 0 & 0 & 0 \end{bmatrix} \tag{3.13}$$

The subscript n in both Equation (3.12) and Equation (3.13) denotes the number of rows and columns of the matrix, i.e., they are both $n \times n$ square matrices.

3.3.6 Hermitian Matrix

A Hermitian matrix is a matrix which by definition is equal to its Hermitian adjoint [29, p.169], which is covered in Section 3.3.3. If we define $\mathbf{A} = a_{ij}$ as an $m \times m$ matrix, then we know that a Hermitian matrix must fulfill

$$\mathbf{A} = \mathbf{A}^H. \tag{3.14}$$

A general Hermitian matrix can be written as

$$\begin{bmatrix} r_1 & c_1 & c_2 & \dots & c_m \\ \overline{c_1} & r_2 & c_{m+1} & \dots & \vdots \\ \overline{c_2} & \overline{c_{m+1}} & r_2 & & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \overline{c_m} & \dots & \dots & \dots & r_m \end{bmatrix}, \tag{3.15}$$

where r_n denotes a real-valued number, c_n denotes a complex-valued number, and $\overline{c_n}$ denotes its complex conjugate. From Equation (3.15), we observe that a Hermitian matrix must have a real-valued diagonal, and that all other values can be complex with its complex conjugate mirrored on the diagonal.

3.3.7 Persymmetric Matrix

A persymmetric matrix is a square matrix that is symmetric to its southwest-to-northeast diagonal. If we define $\mathbf{A} = a_{ij}$ as an $m \times m$ matrix, then a persymmetric matrix must fulfill

$$a_{ij} = a_{m-j+1,m-i+1}, \tag{3.16}$$

which is the equivalent to requiring $\mathbf{A} = \mathbf{J}\mathbf{A}^T\mathbf{J}$ [30, p. 193].

A visual representation of a 4×4 persymmetric matrix is shown in Figure 3.3, and an example of a persymmetric matrix is shown in Equation (3.17).

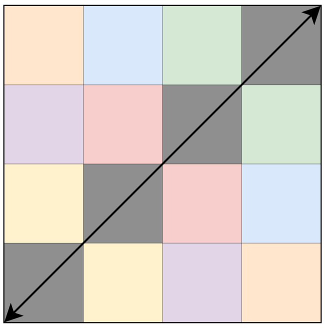


Figure 3.3: Visual representation of a persymmetric matrix. Equal colors indicate equal values. The gray cells indicate the southwest-to-northeast diagonal. The values along this diagonal are not relevant for the Persymmetric property, and they are not required to be equal to each other.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 4 & 3 & 3 \\ 6 & 2 & 4 & 2 \\ 1 & 6 & 5 & 1 \end{bmatrix} \tag{3.17}$$

3.3.8 Unitary Matrix

A unitary matrix, $\mathbf{U}$, is a matrix that fulfills the property presented in Equation (3.18) [29, p. 67].

$$\mathbf{U}^H\mathbf{U} = \mathbf{I} \tag{3.18}$$

Equation (3.18) implies that $\mathbf{U}$ is a square matrix.

3.3.9 Matrix Multiplication

If $\mathbf{A}$ is an $n \times m$ matrix and $\mathbf{B}$ is an $m \times p$ matrix, then the matrix multiplication, $\mathbf{C} = \mathbf{AB}$ is the $n \times p$ matrix where [29, p. 7]

$$c_{ij} = \sum_{k=1}^{m} a_{ik}b_{kj}. \tag{3.19}$$

From Equation (3.19), we observe that the number of columns in $\mathbf{A}$ must be equal to the number of rows in $\mathbf{B}$.

3.3.10 Euclidean Norm

The Euclidean norm, denoted with the $|| \cdot ||_2$-operator, is defined as

$$||\mathbf{A}||_2 = \sqrt{\sum_{i=1}^{n} |a_i|^2}, \tag{3.20}$$

where $\mathbf{A} = a_i$ is the n-dimensional vector [29, p. 291].

Example of Euclidean Norm

Let $\mathbf{A} = [1, 3, 5, 7, 9]$. Then

$$||\mathbf{A}||_2 = \sqrt{1^2 + 3^2 + 5^2 + 7^2 + 9^2} = 12.85.$$

3.4 In-Phase and Quadrature signals

Received RF signals are commonly represented as a complex numbers referred to as In-Phase and Quadrature (IQ) signals [31]. The In-phase component is represented as a real-valued number while the quadrature component is imaginary-valued. This means that each IQ value can be represented as a complex value. A representation of an IQ value is shown in Figure 3.4.

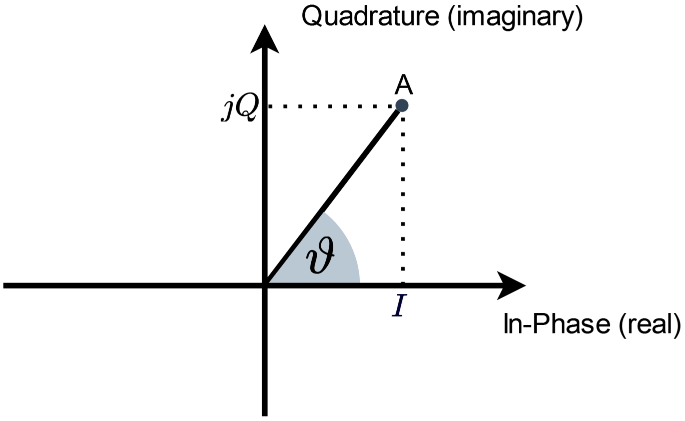


Figure 3.4: IQ value in the complex plane.

An IQ value contains information of both the signal amplitude, A, and the phase, ϑ. The two values can be calculated as shown in Equation (3.21a) and Equation (3.21b), respectively.

$$A = \sqrt{I^2 + Q^2} \tag{3.21a}$$

$$\vartheta = \arctan\left(\frac{Q}{I}\right) \tag{3.21b}$$

From Equation (2.2), we remember that the AoA can be estimated between two antennas by using the phase difference, ϕ, between the sampled values. If Equation (3.21b) is used for obtaining the two phases, ϑ_1 and ϑ_2, then $\phi = \vartheta_2 - \vartheta_1$.

3.5 Azimuth and Elevation

In 3D, the AoA is often described using the two angles, elevation (θ) and azimuth (φ). The two angles are visualized in Figure 3.5. From this figure, we observe that elevation describes the direction in z-space. Usually, when working with AoA, the valid range for θ is $[0°, 90°]$. $\theta = 0$ indicates that the AoA is located somewhere along the z-axis and $x = y = 0$. $\theta = 90°$ indicates that the AoA is located in the x, y-plane only and in the same height as the receiving array. The valid range for azimuth is usually [0°,360°], and this angle describes the rotation in the x, y-plane. From these definitions, we observe that we cannot achieve a scenario where $\theta = 0°$ and $\varphi \neq 0°$.

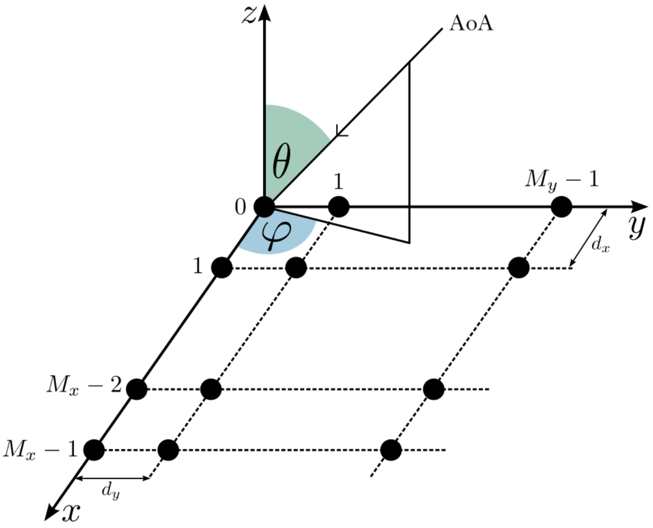


Figure 3.5: Layout of an $M_x \times M_y$ Uniform Rectangular Array (URA).

3.6 The Multiple Signal Classification Algorithm

The MUSIC algorithm was presented by Schmidt in 1985 [19], and is to this day a popular algorithm, known for its great performance. The algorithm has multiple use cases, where estimation of AoA is one of them [19]. The MUSIC algorithm is based on extracting the signal data from the estimated eigenvectors of the received signal, and to perform a search within a specific region for possible AoAs. This region is often limited to the values for θ and φ as described in Section 3.5. The MUSIC search function is based on detecting the K sets of AoA, i.e., $(\theta, \varphi) \triangleq [(\theta_1, \varphi_1), (\theta_2, \varphi_2), ..., (\theta_K, \varphi_K)]$, which yields the K-highest correlations between the received signal information and the receiving antenna array. This section first presents the data model for the algorithm before presenting a real-valued (RV) MUSIC approach for the algorithm. In order to understand how the RV-MUSIC algorithm works for a Uniform Rectangular Array (URA), an introduction to the RV-MUSIC algorithm for a Uniform Linear Array (ULA), presented in [24], is given.

The following section uses letters to symbolize different quantities used in the MUSIC algorithm. Table 3.1 holds all quantities with a description.

Table 3.1: Descriptions of the letters used to symbolize different quantities.

Letter	Description
K	Number of incoming sources
D	Total number of elements in the antenna array
M_x, M_y	Number of antenna elements in x and y direction, respectively
N	Number of snapshots
d_x, d_y	distance between antenna-elements in x or y direction, respectively

3.6.1 Data model

Consider a scenario similar to the one in Figure 3.5. Let K denote the number of incoming signals with unknown AoA, and let $D = M_x \cdot M_y$ be the total number of elements in the antenna array. The elements are spaced with a distance d_x in the x-direction and d_y in the

y-direction as shown in Figure 3.5. This structure indicates that it is a URA. If N snapshots are received on each element as presented in Equation (2.3), then the generic data model for the received signal values at a single snapshot t, $\mathbf{X}(t)$, forms the $D \times 1$ matrix presented in Equation (3.22).

$$\mathbf{X}(t) = \mathbf{A}\mathbf{s}(t) + \mathbf{n}(t)\,,\ t = 1, 2, ..., N \tag{3.22}$$

The matrix $\mathbf{A}$ at snapshot t is a $D \times K$ matrix of signal direction vectors. This matrix can be represented as

$$\mathbf{A} = [\mathbf{a}(\theta_1, \varphi_1), \mathbf{a}(\theta_2, \varphi_2), ..., \mathbf{a}(\theta_K, \varphi_K)], \tag{3.23}$$

where $\mathbf{a}(\theta_k, \varphi_k)$ is the $D \times 1$ steering vector. The k-subscript denotes one specific AoA out of the K present AoAs, i.e., $1 \leq k \leq K$. Before defining the steering vector for a URA, we observe the steering vectors in x- and y-directions separately [7]:

$$\begin{aligned}\mathbf{a}_x(\theta_k, \varphi_k) = \mathbf{a}_x(\omega) &= \begin{bmatrix} \exp\left(j\frac{2\pi d_x \cdot 0}{\lambda}\omega\right) \\ \exp\left(j\frac{2\pi d_x \cdot 1}{\lambda}\omega\right) \\ \vdots \\ \exp\left(j\frac{2\pi d_x \cdot (M_x-1)}{\lambda}\omega\right) \end{bmatrix} \\ &= [\alpha^0, \alpha^1, ..., \alpha^{M_x-1}]^T \end{aligned} \tag{3.24}$$

$$\begin{aligned}\mathbf{a}_y(\theta_k, \varphi_k) = \mathbf{a}_y(\psi) &= \begin{bmatrix} \exp\left(j\frac{2\pi d_y \cdot 0}{\lambda}\psi\right) \\ \exp\left(j\frac{2\pi d_y \cdot 1}{\lambda}\psi\right) \\ \vdots \\ \exp\left(j\frac{2\pi d_y \cdot (M_y-1)}{\lambda}\psi\right) \end{bmatrix} \\ &= [\beta^0, \beta^1, ..., \beta^{M_y-1}]^T. \end{aligned} \tag{3.25}$$

From Equation (3.24) and Equation (3.25), we can observe two properties. The first being that $d_x \cdot (M_x - 1)$ describes the location of the M_x^{th} antenna element in the x-direction. This also applies for the steering vector in the y-direction. For ULA and URA, the antenna elements in x- and y-direction are spaced with distances of d_x and d_y, respectively. However, this also means that for non-uniform arrays, the product $(d_x \cdot m)$, where $(0 \leq m \leq M_x - 1)$ can be replaced with the exact location for the selected antenna element in the x-direction. The same goes for the antenna elements in the $y-$direction. The second observation is that $\alpha = \exp\left(j\frac{2\pi d_x}{\lambda}\omega\right)$ and $\beta = \exp\left(j\frac{2\pi d_y}{\lambda}\psi\right)$, where we define the two signal components $\omega(\theta, \varphi) = \sin(\theta)\cos(\varphi)$ and $\psi(\theta, \varphi) = \sin(\theta)\sin(\varphi)$. The steering vector for a URA can then be obtained by using the Kronecker product, defined in Section 3.3.4, between the steering vectors in x- and y-direction:

$$\begin{aligned}\mathbf{a}(\theta_k,\varphi_k) =&\mathbf{a}_x(\omega)\otimes\mathbf{a}_y(\psi)\\ =&[\alpha^0\beta^0,\alpha^0\beta^1,...,\alpha^0\beta^{M_y-1},\\ &\alpha^1\beta^0,\alpha^1\beta^1,...,\alpha^1\beta^{M_y-1},\\ &\qquad\vdots\\ &\alpha^{M_x-1}\beta^0,\alpha^{M_x-1}\beta^1,...,\alpha^{M_x-1}\beta^{M_y-1}]^T.\end{aligned} \tag{3.26}$$

$\mathbf{a}(\theta,\varphi)$ is therefore the $M_xM_y \times 1$ steering vector. By performing the Kronecker product between the steering vector in x- and y-direction, we get a resulting steering vector where all combinations x- and y- coordinates are included. A visual representation of this is shown in Figure 3.6, where $M_x = M_y = 4$. Remember that the resulting vector is one-dimensional, while it keeps information on the antenna structure in two dimensions.

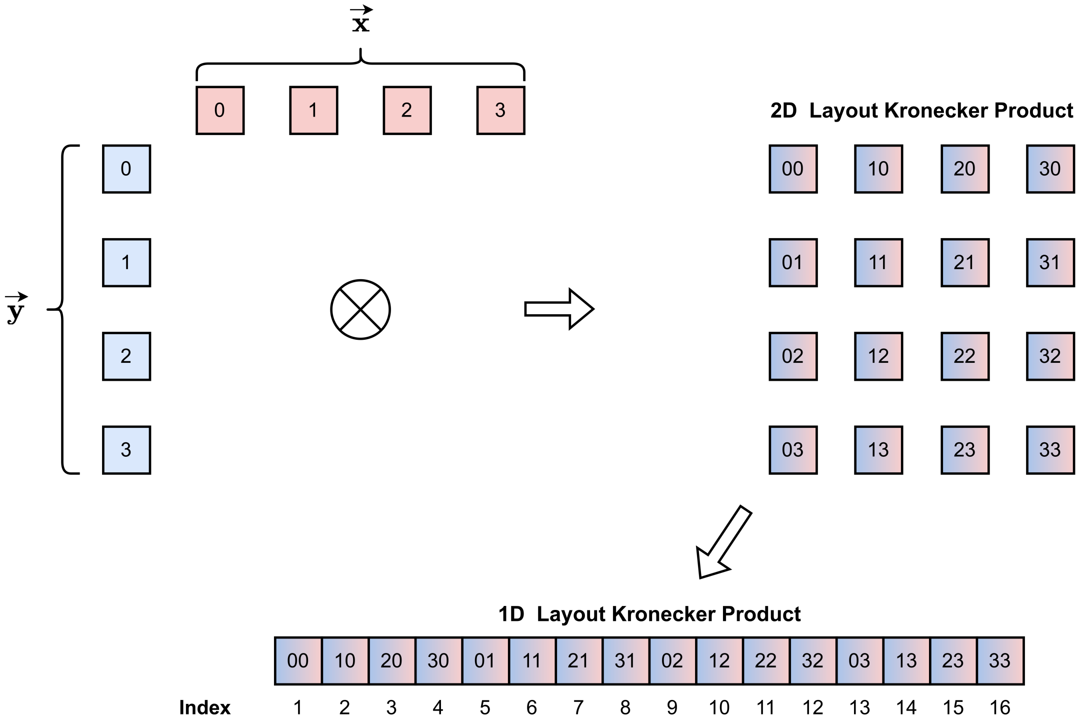


Figure 3.6: Visual representation of how the Kronecker product can be used for obtaining the steering vector for a URA.

$\mathbf{s}(t)$ in Equation (3.22) is the $K \times 1$ source waveforms vector, while $\mathbf{n}(t)$ is the $D \times 1$ vector of white noise [19, 32].

Including all snapshots, we have the following equation and matrices with the dimensions presented in Table 3.2 for the MUSIC algorithm data model.

$$\mathbf{X} = \mathbf{AS} + \mathbf{N} \tag{3.27}$$

Table 3.2: MUSIC matrix dimensions.

Matrix	Description
$\mathbf{X}$	$D \times N$
$\mathbf{A}$	$D \times K$
$\mathbf{S}$	$K \times N$
$\mathbf{N}$	$D \times N$

3.6.2 Covariance matrix

The covariance matrix of the received signals can be expressed as

$$\mathbf{R}_{xx} = E\left[\mathbf{X}(t)\mathbf{X}^H(t)\right] = \mathbf{A}\mathbf{R}_S\mathbf{A}^H + \sigma_N^2\mathbf{I}, \tag{3.28}$$

where $\mathbf{R}_S$ is the signal covariance, σ_N^2 is the noise variance, and $\mathbf{I}$ is the identity matrix [19, eq. 2]. $\mathbf{R}_{xx}$ has D eigenvalues $\lambda_1 \geq \lambda_2 ... \geq \lambda_K \geq \lambda_{K+1} ... \geq \lambda_D = \sigma^2$ and corresponding eigenvectors [19]. The eigenvectors, $\mathbf{E}$ shown in Equation (3.29), can be divided into signal subspace, $\mathbf{E_S} = \{e_1, e_2, .., e_K\}$, and noise subspace, $\mathbf{E_N} = \{e_{K+1}, e_{K+2}, .., e_D\}$.

$$\mathbf{E} = \begin{array}{cc}
\begin{array}{cccccccc} 1 & 2 & ... & K & K{+}1 & K{+}2 & ... & D \end{array} & \\
\left[\begin{array}{cccccccc}
e_{(1,1)} & e_{(1,2)} & \cdots & e_{(1,K)} & e_{(1,K+1)} & e_{(1,K+2)} & \cdots & e_{(1,D)} \\
e_{(2,1)} & e_{(2,2)} & \cdots & e_{(2,K)} & e_{(2,K+1)} & e_{(2,K+2)} & \cdots & e_{(2,D)} \\
e_{(3,1)} & e_{(3,2)} & \cdots & e_{(3,K)} & e_{(3,K+1)} & e_{(3,K+2)} & \cdots & e_{(3,D)} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
e_{(D,1)} & e_{(D,2)} & \cdots & e_{(D,K)} & e_{(D,K+1)} & e_{(D,K+2)} & \cdots & e_{(D,D)}
\end{array}\right] & \begin{array}{c} 1 \\ 2 \\ 3 \\ \vdots \\ D \end{array} \\
\underbrace{\qquad\qquad\qquad\qquad\qquad}_{\mathbf{E}_S}\;\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{\mathbf{E}_N} &
\end{array} \tag{3.29}$$

In practical applications, the maximum likelihood estimator of $\mathbf{R}_{xx}$, $\hat{\mathbf{R}}_{xx}$ can be expressed as [33]

$$\hat{\mathbf{R}}_{xx} = \frac{1}{N}\sum_{t=1}^{N}\mathbf{x}(t)\mathbf{x}^H(t). \tag{3.30}$$

$\hat{\mathbf{R}}_{xx}$ will always be a Hermitian matrix. *Proof:*

$$(\mathbf{A}\mathbf{A}^H)^H = (\mathbf{A}^H)^H\mathbf{A}^H = \mathbf{A}\mathbf{A}^H \tag{3.31}$$

The first equality follows from the reverse-order law presented in Equation (3.8), while the last equality is due to the property that is presented in Equation (3.9). Equation (3.31) proofs that the resulting matrix has the property presented in Equation (3.14), hence the matrix is Hermitian.

3.6.3 Search Function

The MUSIC algorithm supports two search functions, depending on which subspace, $\mathbf{E}_N/\mathbf{E}_S$, is being used [19]. The goal is to find the K peaks of P_{MU} in Equation (3.32).

$$\text{Signal subspace: } P_{\text{MU}}(\theta,\varphi) = \mathbf{a}^H(\theta,\varphi)\mathbf{E_S}\mathbf{E_S}^H\mathbf{a}(\theta,\varphi) \tag{3.32a}$$

$$\text{Noise subspace: } P_{\text{MU}}(\theta,\varphi) = \frac{1}{\mathbf{a}^H(\theta,\varphi)\mathbf{E_N}\mathbf{E_N}^H\mathbf{a}(\theta,\varphi)}. \tag{3.32b}$$

Equation (3.32) can be further simplified to

$$\text{Signal subspace: } \max_{\theta,\varphi}\ P_{\text{MU}}(\theta,\varphi) = \mathbf{a}^H(\theta,\varphi)\mathbf{E_S}\mathbf{E_S}^H\mathbf{a}(\theta,\varphi) \tag{3.33a}$$

$$\text{Noise subspace: } \min_{\theta,\varphi}\ P_{\text{MU}}(\theta,\varphi) = \mathbf{a}^H(\theta,\varphi)\mathbf{E_N}\mathbf{E_N}^H\mathbf{a}(\theta,\varphi) \tag{3.33b}$$

Equation (3.33) states that the goal is to find the K sets of $(\theta,\varphi) \triangleq [(\theta_1,\varphi_1),(\theta_2,\varphi_2),...,(\theta_K,\varphi_K)]$, which gives the maximum or minimum values for P_{MU}, depending on the selected subspace. The search function can be described in simple terms. We know that the steering vector $\tilde{\mathbf{a}}(\theta,\varphi)$ holds information on the receiving antenna structure, $\mathbf{E}_S$ holds information on the received signal, while $\mathbf{E}_N$ holds information on the received noise. We can choose if we want to find the AoA by observing where the correlation between the receiving antenna strucutre and the signal information is the highest, or where the antenna structure correlates the least with the noise information.

3.7 A Real-Valued MUSIC Algorithm

$\mathbf{X}$, in Equation (3.27), is a complex matrix with a complex-valued covariance matrix, which yields complex-valued eigenvectors. From Equation (3.29), it follows that both $\mathbf{E}_S$ and $\mathbf{E}_N$ will be complex-valued. The steering vector $\mathbf{a}(\varphi,\theta)$ in Equation (3.26) is also complex-valued, which then means that Equation (3.33) includes complex-valued arithmetic. From Equation (3.1), we observe that complex-valued arithmetic has a higher complexity compared to real-valued arithmetic. Multiple available papers, such as [7, 24, 25], discuss this problem and provide a possible solution by transforming the complex-valued matrices to real-valued matrices using *unitary transformation*. The following section is used to present the theory for this approach.

Keh-Chiarng Huarng and Chien-Chung Yeh propose to utilize a unitary transformation method to convert the complex matrices, $\hat{\mathbf{R}}_{xx}$ and $\mathbf{a}(\theta,\varphi)$, to real-valued matrices [24]. This results in real-valued eigenvectors which results in the P_{MU} only containing real-valued arithmetic, hence lowering the computation complexity. The unitary transformation requires the matrices to be transformed to have Hermitian and Persymmetric properties. From Equation (3.31), we know that $\hat{\mathbf{R}}_{xx}$ is Hermitian but not necessarily Persymmetric. The persymmetric property can be obtained by forward/backward (FB) averaging $\hat{\mathbf{R}}_{xx}$, as done in [22] and [34]:

$$\hat{\mathbf{R}}_{\text{FB}} = \frac{1}{2}\left(\mathbf{R} + \mathbf{J}\mathbf{R}^*\mathbf{J}\right). \tag{3.34}$$

$\hat{\mathbf{R}}_{\text{FB}}$ is now transformed to have both Persymmetric and Hermitian properties [34]. The unitary transformation method,

$$\mathbb{R}_{xx} = \mathbf{U}\hat{\mathbf{R}}_{\text{FB}}\mathbf{U}^H, \tag{3.35}$$

where $\mathbf{U}$ is a unitary matrix, can then be applied to obtain the real-valued covariance matrix, $\mathbb{R}_{xx}$.

From [24], the unitary matrix is provided for a Uniform Linear Array (ULA). If D denotes the number of elements in the linear array, then

$$\mathbf{U} = \frac{1}{\sqrt{2}}\begin{bmatrix} \mathbf{I}_{D/2} & \mathbf{J}_{D/2} \\ j\mathbf{J}_{D/2} & -j\mathbf{I}_{D/2} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & \cdots & \cdots & D \end{array} \\ \left[\begin{array}{ccccc} 1 & 0 & \cdots & 0 & 1 \\ 0 & 1 & & 1 & 0 \\ \vdots & & \ddots \; \cdot^{\cdot^{\cdot}} & & \vdots \\ \vdots & & \cdot^{\cdot^{\cdot}} \; \ddots & & \vdots \\ 0 & j & & -j & 0 \\ j & 0 & \cdots & 0 & -j \end{array}\right] \begin{array}{c} 1 \\ 2 \\ \vdots \\ \vdots \\ \vdots \\ D \end{array} \end{array}. \tag{3.36}$$

Real-valued eigenvalues and eigenvectors can then be found from $\mathbb{R}_{xx}$, and the real-valued signal- and noise- subspace, denoted with $\mathbb{E}_S$ and $\mathbb{E}_N$, respectively, can then be formed similarly as presented in Equation (3.29). In order to find the corresponding unitary matrix for a URA, the Kronecker product is once more used. We define

$$\mathbf{U}_x = \frac{1}{\sqrt{2}}\begin{bmatrix} \mathbf{I}_{M_x/2} & \mathbf{J}_{M_x/2} \\ j\mathbf{J}_{M_x/2} & -j\mathbf{I}_{M_x/2} \end{bmatrix}, \tag{3.37}$$

and

$$\mathbf{U}_y = \frac{1}{\sqrt{2}}\begin{bmatrix} \mathbf{I}_{M_y/2} & \mathbf{J}_{M_y/2} \\ j\mathbf{J}_{M_y/2} & -j\mathbf{I}_{M_y/2} \end{bmatrix}. \tag{3.38}$$

The unitary matrix for a URA, $\mathbf{T}$, is then the $(M_xM_y \times M_xM_y)$ matrix defined by [7]

$$\mathbf{T} = \mathbf{U}_x \otimes \mathbf{U}_y. \tag{3.39}$$

The real-valued covariance matrix is then found from

$$\mathbb{R}_{xx,\text{URA}} = \frac{1}{2}(\mathbf{T}\hat{\mathbf{R}}_{\text{FB}}\mathbf{T}^H). \tag{3.40}$$

The complex-valued steering vector must also be transformed to a real-valued steering vector, $\tilde{\mathbf{a}}$. From [24], we observe the transformation for a ULA steering vector to be defined as

$$\tilde{\mathbf{a}}_x(\omega) = \mathbf{U}_x e^{-j(M_x-1)\alpha/2}\mathbf{a}_x(\omega), \tag{3.41}$$

for the steering vector in the x-direction, and

$$\tilde{\mathbf{a}}_y(\psi) = \mathbf{U}_y e^{-j(M_y-1)\beta/2}\mathbf{a}_y(\psi) \tag{3.42}$$

for the y-direction.

The complete real-valued steering vector for a URA can then be obtained by

$$\tilde{\mathbf{a}}(\theta, \varphi) = \tilde{\mathbf{a}}_x(\omega) \otimes \tilde{\mathbf{a}}_y(\psi). \tag{3.43}$$

We can now replace the complex-valued matrices and vectors in Equation (3.33) with the obtained real-valued matrices and vectors. The real-valued MUSIC search function then becomes

$$\text{Signal subspace: } \max_{\theta,\varphi}\ \mathbb{P}_{\text{MU}}(\theta, \varphi) = [\tilde{\mathbf{a}}_x(\omega) \otimes \tilde{\mathbf{a}}_y(\psi)]^H \mathbb{E}_S \mathbb{E}_S^H [\tilde{\mathbf{a}}_x(\omega) \otimes \tilde{\mathbf{a}}_y(\psi)] \tag{3.44a}$$

$$\text{Noise subspace: } \min_{\varphi,\psi}\ \mathbb{P}_{\text{MU}}(\varphi, \psi) = [\tilde{\mathbf{a}}_x(\omega) \otimes \tilde{\mathbf{a}}_y(\psi)]^H \mathbb{E}_S \mathbb{E}_S^H [\tilde{\mathbf{a}}_x(\omega) \otimes \tilde{\mathbf{a}}_y(\psi)]. \tag{3.44b}$$

Chapter 4

Derivation of application-specific MUSIC algorithm

In Chapter 3, two versions of the search function for the MUSIC algorithm are presented: one complex-valued (CV) and one real-valued (RV). The following chapter discusses the advantages and disadvantages of the two versions before providing a possible modification to the results from [24] and [7] for optimizing the MUSIC algorithm for the specific system presented in Section 2.1.

4.1 Antenna Layout

The layout of the antenna structure is essential for further explanation of the matrices and design choices presented in this chapter. For receiving and sampling the incoming CTE from the transmitting tag, the ISP1807-AoA-DK [35] locator PCB is used. It has a square layout with $D = 12$ antenna elements. The layout of the array is shown in Figure 4.1. The figure indicates that the used antenna array is not a standard URA or ULA structure, meaning that the equations presented in Section 3.6 must therefore be modified to fit the presented layout.

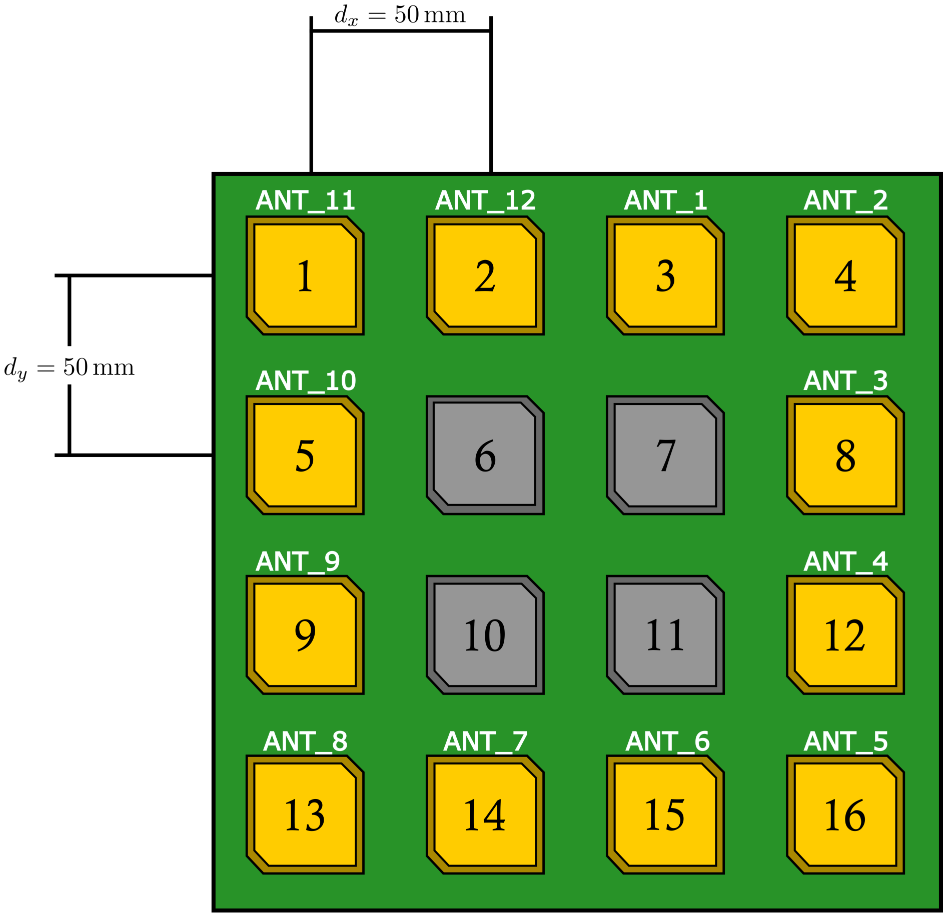


Figure 4.1: Layout of the antenna elements. Each yellow element indicates one antenna element, while the gray elements indicate the non-existing elements. These elements are used for deriving the final solution of the RV-MUSIC algorithm.

4.2 Discussion of CV and RV MUSIC Algorithm

From the presentation of BLE and how it can be used in direction finding applications, presented in Section 2.2, we know that there should be only one source for the sampled CTE, hence $K = 1$. In Section 2.1, it is described how the CTE is received together with the corresponding ID of the transmitting tag, and the results from the AoA estimations is to be visualized for the correct tag ID. A scenario where two transmitters transmits the CTE at the same time can occur, and in those cases, the sampled signal will contain combined information from the two sources. In such scenarios, using $K = 1$ can result in a wrongly estimated AoA, as the search function can find greater correlation for the wrong signal. However, the system is not able to detect multiple tags from one CTE sample, as it is unable to assign the other, unknown IDs.

Multipath propagation [36] is also a well-known source of error, where the transmitted signal typically bounces on the walls, floors and ceilings of the closed structure. In these cases, it is hard to catch such errors when only searching for one peak. However, searching for multiple peaks when only one transmitter is present or no multipath propagation is present can also introduce errors from the estimations or at least more uncertainty. A decision of only searching for one peak is therefore taken, as this also significantly reduces the complexity of the search function presented in Equation (3.33). To reduce the possible errors from multiple CTE sources or multipath propagation, one can apply post filtering of the AoA found by the algorithm. Filtering is not included in the objectives for this thesis, and will not be discussed.

From the above discussion and decision of searching for only one peak, we can now quantize the values presented in Table 3.1. From Section 2.3, the number of snapshots, N, is calculated to 3, given the CTE configurations presented in Section 2.1. The quantized values are further used in deriving the application-specific MUSIC algorithm, allowing the hardware implementation, presented in Chapter 5, to be optimized for the specific use case. The quantized values are presented in Table 4.1.

Table 4.1: Application specific values for the MUSIC algorithm.

Quantity	Value
K	1
D	12
M_x, M_y	4
N	3
$d = d_x = d_y$	50 mm

The additional operations required for the RV-MUSIC algorithm are FB-averaging and unitary transformation, both presented in Section 3.7. From the values in Table 4.1, we know that $\mathbf{X}$ is a 12×3 $(D \times N)$ matrix, and $\hat{\mathbf{R}}_{xx}$ in Equation (3.30) is therefore a 12×12 complex-valued matrix. This also means that $\mathbf{J}$ is the 12×12 exchange matrix.

FB averaging requires two matrix multiplication- operands and one matrix addition while the unitary transformation requires two matrix multiplications. From Equation (3.19), we can observe that a matrix multiplication of two $m \times m$ matrices requires m^3 multiplications of the elements. All matrices for both of the required operations are complex-valued and have dimensions 12×12. From Equation (3.1), we know that a complex multiplication requires four real-valued multiplications. An estimation of the total number of real-valued multiplications for the transformation operations is provided in Table 4.2.

Table 4.2: Operations required for RV-MUSIC transformation. The calculated values in this table are found from the dimensions of the matrices included in Equation (3.34) and Equation (3.35).

Operation	Number of real-valued multiplications
FB averaging	$(12^3 \cdot 2) \cdot 4 = 13824$
Unitary transformation	$(12^3 \cdot 2) \cdot 4 = 13824$
Total	$13824 + 13824 = 27648$

The benefit of performing the unitary transformation and obtaining the real-valued eigenvectors and steering vector must also be discussed. From Section 3.6, we know that the goal is to search through P_{MU} in Equation (3.33) or $\mathbb{P}_{\mathrm{MU}}$ in Equation (3.44). With $K = 1$, the relevant vectors will have the dimensions presented in Table 4.3. From these dimensions, it is easily observable from Equation (3.32) that searching through the signal subspace will significantly reduce the number of operations required.

Table 4.3: RV-MUSIC search function matrix dimensions.

Matrix	Dimensions
$\mathbf{a}^H(\theta, \varphi)$	1×12
$\mathbf{E}_S$	12×1
$\mathbf{E}_N$	12×11

Using the definition of matrix multiplication in Equation (3.19), we observe that the multiplication of $\mathbf{a}^H(\theta, \varphi)\mathbf{E}_S$ requires 48 multiplications if the vectors are complex-valued, and 12 multiplications if the vectors are real-valued. The difference is therefore $48 - 12 = 36$ more multiplications for each iteration of the search. From Table 4.2, we remember that the required number of multiplications for the real-valued transformation is 27 648. This means that $27648/36 = 768$ complex-valued vector multiplication requires the same amount of multiplications as the transformation operations. The number of iterations needed for the search function is further discussed in Chapter 5, and is originally far greater than 768. Using complex-valued vectors also limit the level of parallelism of the implementation. For every complex-valued vector multiplication, one could perform 4 parallel real-valued vector multiplications. A more detailed comparison of possible architectures for the two versions of the MUSIC algorithm is given in Section 7.3.

Another important factor to discuss when comparing the real-valued vs. complex-valued approach is how this decision influences the results of the implemented design by Tommy A. Opstad [1]. From his thesis, we also observe that the real-valued approach is beneficial, as it allows for a greater level of parallelism [1].

To summarize, implementing a real-valued MUSIC algorithm requires an extra step compared to the complex-valued version, resulting in an additional step to the algorithm. However, implementing a real-valued version will most likely be beneficial as it entails a simpler hardware architecture that allows for greater level of parallelism and flexibility. The real-valued approach is therefore chosen as the preferred method of implementation.

4.3 Derivation

For deriving the specific RV MUSIC algorithm for the presented antenna structure in Figure 4.1, the strategy is to first use the proposed approach for a standard URA structure, presented in Section 3.7, before making further modifications. These modifications are presented in the following section.

From Figure 2.8, we observe that we expect 37 samples when using 2 µs sample- and switch slots and a CTE length of 160µs. The received I/Q samples are aligned to a 12×3 matrix in the following form

$$\mathbf{X} = \begin{array}{c} \left[\begin{array}{ccc} \text{Sample 1} & \text{Sample 13} & \text{Sample 25} \\ \text{Sample 2} & \text{Sample 14} & \text{Sample 26} \\ \text{Sample 3} & \text{Sample 15} & \text{Sample 27} \\ \text{Sample 4} & \text{Sample 16} & \text{Sample 28} \\ \text{Sample 5} & \text{Sample 17} & \text{Sample 29} \\ \text{Sample 6} & \text{Sample 18} & \text{Sample 30} \\ \text{Sample 7} & \text{Sample 19} & \text{Sample 31} \\ \text{Sample 8} & \text{Sample 20} & \text{Sample 32} \\ \text{Sample 9} & \text{Sample 21} & \text{Sample 33} \\ \text{Sample 10} & \text{Sample 22} & \text{Sample 34} \\ \text{Sample 11} & \text{Sample 23} & \text{Sample 35} \\ \text{Sample 12} & \text{Sample 24} & \text{Sample 36} \end{array}\right] \\ \begin{array}{ccc} \underbrace{\hphantom{\text{Sample 12}}}_{\text{Snapshot 1}} & \underbrace{\hphantom{\text{Sample 24}}}_{\text{Snapshot 2}} & \underbrace{\hphantom{\text{Sample 36}}}_{\text{Snapshot 3}} \end{array} \end{array} \begin{array}{l} \text{ANT\_11} \\ \text{ANT\_12} \\ \text{ANT\_1} \\ \text{ANT\_2} \\ \text{ANT\_10} \\ \text{ANT\_3} \\ \text{ANT\_9} \\ \text{ANT\_4} \\ \text{ANT\_8} \\ \text{ANT\_7} \\ \text{ANT\_6} \\ \text{ANT\_5} \\ \\ \end{array} . \tag{4.1}$$

The pattern for sampling the values was decided in the project thesis [8]. In this project, it was decided to sample each row from left to right starting in the top left corner. The order for the samples will not influence the performance of the algorithm.

The covariance matrix is then found using Equation (3.30), and from Section 3.6.2 we remember that $\hat{\mathbf{R}}_{xx}$ is a Hermitian matrix. This matrix needs to be transformed to a Hermitian, Persymmetric matrix by FB averaging as presented in Equation (3.34). The resulting matrix, $\hat{\mathbf{R}}_{\text{FB}}$, is a Hermitian, Persymmetric matrix that can be transformed to a real-valued matrix using unitary transformation. The unitary matrix, $\mathbf{U}$, is found by modifying the unitary matrix $\mathbf{T} = \mathbf{U}_x \otimes \mathbf{U}_y$. The reason for the need of a modification is that this matrix is defined for a URA while the actual layout, presented in Figure 4.1, has a non-URA layout.

4.3.1 Modifying the Kronecker Product

An illustration of how the Kronecker product can be used for obtaining information of a URA steering vector is presented in Figure 3.6. Remembering that the chosen antenna layout in Figure 4.1 can be seen as a URA structure without the four middle components, an approach of removing the non-existing elements from the Kronecker product can be done for obtaining the correct values. The illustration in Figure 3.6 is further modified to highlight the non-existing elements. The updated illustration is presented in Figure 4.2.

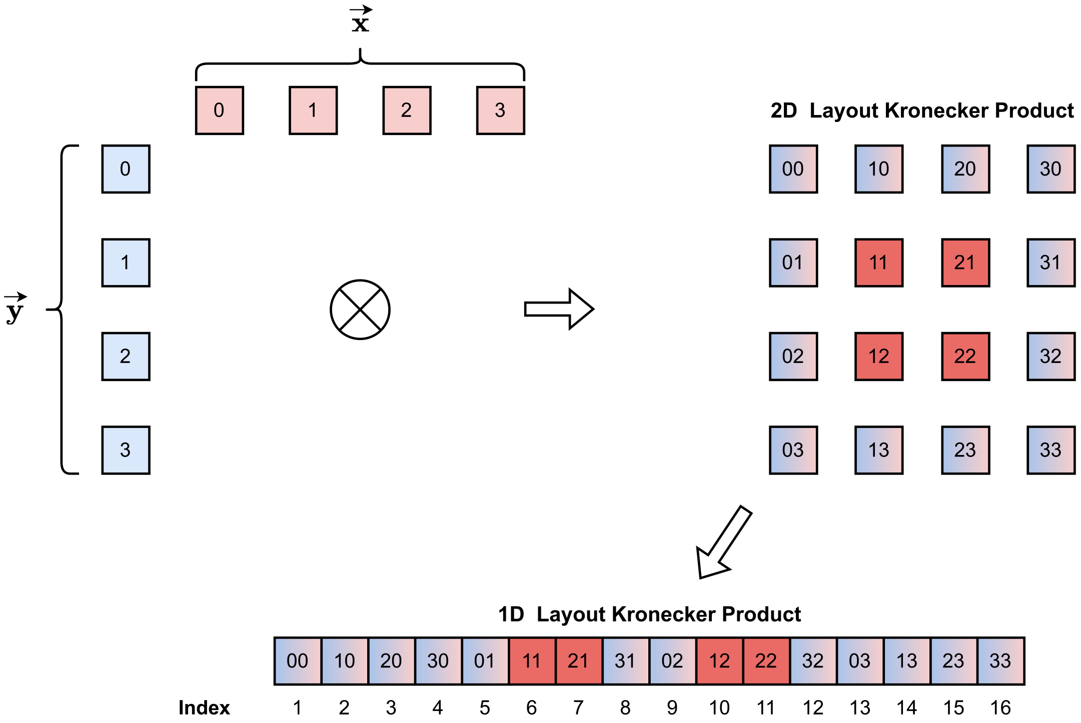


Figure 4.2: Visualization of the Kronecker product.

We observe that the 2D layout of the Kronecker product in Figure 4.2 is very similar to the antenna layout shown in Figure 4.1. By removing the four elements in the middle (marked with red) from the resulting vector, we obtain the correct vector for our antenna layout. If we map the 2D layout to a 1D layout, we observe that the elements with index 6,7,10 and 11 are non-existing elements in our array. They should therefore be excluded from the Kronecker products obtained in the real-valued transformation presented in Section 3.7.

From Section 3.7, we remember that the unitary transformation requires two Kronecker products. The first one is for obtaining the unitary matrix, $\mathbf{T} = \mathbf{U}_x \otimes \mathbf{U}_y$. Using Equation (3.36) and $M_x = M_y = 4$ yields

$$\mathbf{U}_x = \mathbf{U}_y = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & j & -j & 0 \\ j & 0 & 0 & -j \end{bmatrix}. \tag{4.2}$$

The corresponding unitary matrix for a URA is therefore

$$\mathbf{T} = \mathbf{U}_x \otimes \mathbf{U}_y = \frac{1}{2}\begin{bmatrix} 1\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix} & 0\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix} & 0\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix} & 1\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix} \\ 0\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix} & 1\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix} & 1\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix} & 0\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix} \\ 0\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix} & j\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix} & -j\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix} & 0\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix} \\ j\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix} & 0\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix} & 0\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix} & -j\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix} \end{bmatrix}$$

$$= \frac{1}{2}\begin{bmatrix} 1&0&0&1&0&0&0&0&0&0&0&0&1&0&0&1\\ 0&1&1&0&0&0&0&0&0&0&0&0&0&1&1&0\\ 0&j&-j&0&0&0&0&0&0&0&0&0&0&j&-j&0\\ j&0&0&-j&0&0&0&0&0&0&0&0&j&0&0&-j\\ 0&0&0&0&1&0&0&1&1&0&0&1&0&0&0&0\\ 0&0&0&0&0&1&1&0&0&1&1&0&0&0&0&0\\ 0&0&0&0&0&j&-j&0&0&j&-j&0&0&0&0&0\\ 0&0&0&0&j&0&0&-j&j&0&0&-j&0&0&0&0\\ 0&0&0&0&j&0&0&j&-j&0&0&-j&0&0&0&0\\ 0&0&0&0&0&j&j&0&0&-j&-j&0&0&0&0&0\\ 0&0&0&0&0&-1&1&0&0&1&-1&0&0&0&0&0\\ 0&0&0&0&-1&0&0&1&1&0&0&-1&0&0&0&0\\ j&0&0&j&0&0&0&0&0&0&0&0&-j&0&0&-j\\ 0&j&j&0&0&0&0&0&0&0&0&0&0&-j&-j&0\\ 0&-1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0\\ -1&0&0&1&0&0&0&0&0&0&0&0&1&0&0&-1 \end{bmatrix}. \tag{4.3}$$

Remember that this matrix is a 16×16 ($M_xM_y \times M_xM_y$) matrix, and it needs to be modified to fit the layout in Figure 4.1. By removing columns and rows with index 6,7,10, and 11, we obtain the matrix given in Equation (4.4).

$$
\begin{aligned}
\mathbf{T}_{12} &= \frac{1}{2}\left[\begin{array}{cccccccccccccccc|l}
 & & & & & \leftarrow\text{remove} & \leftarrow\text{remove} & & & \leftarrow\text{remove} & \leftarrow\text{remove} & & & & & & \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \\
0 & j & -j & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & j & -j & 0 & \\
j & 0 & 0 & -j & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & j & 0 & 0 & -j & \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & \leftarrow\text{remove} \\
0 & 0 & 0 & 0 & 0 & j & -j & 0 & 0 & j & -j & 0 & 0 & 0 & 0 & 0 & \leftarrow\text{remove} \\
0 & 0 & 0 & 0 & j & 0 & 0 & -j & j & 0 & 0 & -j & 0 & 0 & 0 & 0 & \\
0 & 0 & 0 & 0 & j & 0 & 0 & j & -j & 0 & 0 & -j & 0 & 0 & 0 & 0 & \\
0 & 0 & 0 & 0 & 0 & j & j & 0 & 0 & -j & -j & 0 & 0 & 0 & 0 & 0 & \leftarrow\text{remove} \\
0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & \leftarrow\text{remove} \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & \\
j & 0 & 0 & j & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -j & 0 & 0 & -j \\
0 & j & j & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -j & -j & 0 & \\
0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & \\
-1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1
\end{array}\right] \\
&= \frac{1}{2}\begin{bmatrix}
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & j & -j & 0 & 0 & 0 & 0 & 0 & 0 & j & -j & 0 \\
j & 0 & 0 & -j & 0 & 0 & 0 & 0 & j & 0 & 0 & -j \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & j & -j & j & -j & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & j & j & -j & -j & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 1 & 1 & -1 & 0 & 0 & 0 & 0 \\
j & 0 & 0 & j & 0 & 0 & 0 & 0 & -j & 0 & 0 & -j \\
0 & j & j & 0 & 0 & 0 & 0 & 0 & 0 & -j & -j & 0 \\
0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\
-1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1
\end{bmatrix}
\end{aligned}
\tag{4.4}
$$

The resulting matrix, $\mathbf{T}_{12}$, still fulfills $\mathbf{T}\mathbf{T}^H = \mathbf{T}^H\mathbf{T} = \mathbf{I}$, hence the matrix is unitary, and we can therefore use this matrix to convert $\hat{\mathbf{R}}_{\text{FB}}$ to the real-valued matrix, $\mathbb{R}_{xx}$. This method works for the given antenna layout due to the fact that the non-existing elements in the antenna array are symmetric around the center of the array. Removing symmetric columns and rows from an already symmetric matrix yields a new, symmetric matrix. This is observed in Equation (4.4).

4.3.2 RV Steering vector

From the quantized values in Table 4.1, we can define

$$
\begin{aligned}
\mathbf{a}_x(\theta,\varphi) = \mathbf{a}_x(\omega) &= [1,\alpha,\alpha^2\alpha^3]^T \\
&= \left[1, \exp\left(j\frac{2\pi d}{\lambda}\omega\right), \exp\left(j2\cdot\frac{2\pi d}{\lambda}\omega\right), \exp\left(j3\cdot\frac{2\pi d}{\lambda}\omega\right)\right]^T \qquad (4.5a) \\
\mathbf{a}_y(\theta,\varphi) = \mathbf{a}_y(\psi) &= [1,\beta,\beta^2\beta^3]^T \\
&= \left[1, \exp\left(j\frac{\pi d}{\lambda}\omega\right), \exp\left(j2\cdot\frac{\pi d}{\lambda}\omega\right), \exp\left(j3\cdot\frac{\pi d}{\lambda}\omega\right)\right]^T. \qquad (4.5b)
\end{aligned}
$$

Using Equation (3.41), we can then obtain the real-valued steering vectors in the x-direction:

$$
\begin{aligned}
\tilde{\mathbf{a}}_{\mathbf{x}}(\omega) &= \alpha^{-\frac{3}{2}}\mathbf{U}_x\mathbf{a}_x \\
&= \exp\left(-j\frac{3}{2}\cdot\frac{\pi d}{\lambda}\omega\right)\frac{1}{\sqrt{2}}\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix}\begin{bmatrix}1\\ \exp\left(j1\cdot\frac{2\pi d}{\lambda}\omega\right)\\ \exp\left(j2\cdot\frac{2\pi d}{\lambda}\omega\right)\\ \exp\left(j3\cdot\frac{2\pi d}{\lambda}\omega\right)\end{bmatrix} \\
&= \frac{1}{\sqrt{2}}\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix}\begin{bmatrix}\exp\left(-j\frac{3}{2}\cdot\frac{\pi d}{\lambda}\omega\right)\\ \exp\left(-j\frac{3}{2}\cdot\frac{\pi d}{\lambda}\omega\right)\exp\left(j1\cdot\frac{2\pi d}{\lambda}\omega\right)\\ \exp\left(-j\frac{3}{2}\cdot\frac{\pi d}{\lambda}\omega\right)\exp\left(j2\cdot\frac{2\pi d}{\lambda}\omega\right)\\ \exp\left(-j\frac{3}{2}\cdot\frac{\pi d}{\lambda}\omega\right)\exp\left(j3\cdot\frac{2\pi d}{\lambda}\omega\right)\end{bmatrix} \\
&= \frac{1}{\sqrt{2}}\begin{bmatrix}1&0&0&1\\0&1&1&0\\0&j&-j&0\\j&0&0&-j\end{bmatrix}\begin{bmatrix}\exp\left(-j\frac{3}{2}\cdot\frac{2\pi d}{\lambda}\omega\right)\\ \exp\left(-j\frac{1}{2}\cdot\frac{2\pi d}{\lambda}\omega\right)\\ \exp\left(j\frac{1}{2}\cdot\frac{2\pi d}{\lambda}\omega\right)\\ \exp\left(j\frac{3}{2}\cdot\frac{2\pi d}{\lambda}\omega\right)\end{bmatrix} \qquad (4.6) \\
&= \frac{1}{\sqrt{2}}\begin{bmatrix}\exp\left(-j\frac{3}{2}\cdot\frac{2\pi d}{\lambda}\omega\right)+\exp\left(j\frac{3}{2}\cdot\frac{2\pi d}{\lambda}\omega\right)\\ \exp\left(-j\frac{1}{2}\cdot\frac{2\pi d}{\lambda}\omega\right)+\exp\left(j\frac{1}{2}\cdot\frac{2\pi d}{\lambda}\omega\right)\\ j\left(\exp\left(-j\frac{1}{2}\cdot\frac{2\pi d}{\lambda}\omega\right)-\exp\left(j\frac{1}{2}\cdot\frac{2\pi d}{\lambda}\omega\right)\right)\\ j\left(\exp\left(-j\frac{3}{2}\cdot\frac{2\pi d}{\lambda}\omega\right)-\exp\left(j\frac{3}{2}\cdot\frac{2\pi d}{\lambda}\omega\right)\right)\end{bmatrix} \\
&= \frac{1}{\sqrt{2}}\begin{bmatrix}2\cos\left(\frac{3}{2}\cdot\frac{2\pi d}{\lambda}\omega\right)\\ 2\cos\left(\frac{1}{2}\cdot\frac{2\pi d}{\lambda}\omega\right)\\ 2\sin\left(\frac{1}{2}\cdot\frac{2\pi d}{\lambda}\omega\right)\\ 2\sin\left(\frac{3}{2}\cdot\frac{2\pi d}{\lambda}\omega\right)\end{bmatrix} = \sqrt{2}\begin{bmatrix}\cos\left(\frac{3\pi d}{\lambda}\omega\right)\\ \cos\left(\frac{\pi d}{\lambda}\omega\right)\\ \sin\left(\frac{\pi d}{\lambda}\omega\right)\\ \sin\left(\frac{3\pi d}{\lambda}\omega\right)\end{bmatrix}.
\end{aligned}
$$

The results are obtained by using Eulers Formula, $\exp(j\varphi) = \cos(\varphi) + j\sin(\varphi)$, and the trigonometric relations, $\cos(-\varphi) = \cos(\varphi)$ and $\sin(-\varphi) = \sin(\varphi)$. Using the exact same approach for the ULA steering vector in y-direction yields

$$\tilde{\mathbf{a}}_y(\psi) = \sqrt{2}\begin{bmatrix} \cos\left(\frac{3\pi d}{\lambda}\psi\right) \\ \cos\left(\frac{\pi d}{\lambda}\psi\right) \\ \sin\left(\frac{\pi d}{\lambda}\psi\right) \\ \sin\left(\frac{3\pi d}{\lambda}\psi\right) \end{bmatrix}. \tag{4.7}$$

The RV steering vector for a 4 × 4 URA can then be found from $\tilde{\mathbf{a}}(\theta,\varphi) = \tilde{\mathbf{a}}_x(\omega) \otimes \tilde{\mathbf{a}}_y(\psi)$. Again, the Kronecker product must be modified to fit the application specific antenna layout:

$$\tilde{\mathbf{a}}(\theta,\varphi) = 2\begin{bmatrix} \cos\left(\frac{3\pi d}{\lambda}\omega\right)\cos\left(\frac{3\pi d}{\lambda}\psi\right) \\ \cos\left(\frac{3\pi d}{\lambda}\omega\right)\cos\left(\frac{\pi d}{\lambda}\psi\right) \\ \cos\left(\frac{3\pi d}{\lambda}\omega\right)\sin\left(\frac{\pi d}{\lambda}\psi\right) \\ \cos\left(\frac{3\pi d}{\lambda}\omega\right)\sin\left(\frac{3\pi d}{\lambda}\psi\right) \\ \cos\left(\frac{\pi d}{\lambda}\omega\right)\cos\left(\frac{3\pi d}{\lambda}\psi\right) \\ \cos\left(\frac{\pi d}{\lambda}\omega\right)\cos\left(\frac{\pi d}{\lambda}\psi\right) \\ \cos\left(\frac{\pi d}{\lambda}\omega\right)\sin\left(\frac{\pi d}{\lambda}\psi\right) \\ \cos\left(\frac{\pi d}{\lambda}\omega\right)\sin\left(\frac{3\pi d}{\lambda}\psi\right) \\ \sin\left(\frac{\pi d}{\lambda}\omega\right)\cos\left(\frac{3\pi d}{\lambda}\psi\right) \\ \sin\left(\frac{\pi d}{\lambda}\omega\right)\cos\left(\frac{\pi d}{\lambda}\psi\right) \\ \sin\left(\frac{\pi d}{\lambda}\omega\right)\sin\left(\frac{\pi d}{\lambda}\psi\right) \\ \sin\left(\frac{\pi d}{\lambda}\omega\right)\sin\left(\frac{3\pi d}{\lambda}\psi\right) \\ \sin\left(\frac{3\pi d}{\lambda}\omega\right)\cos\left(\frac{3\pi d}{\lambda}\psi\right) \\ \sin\left(\frac{3\pi d}{\lambda}\omega\right)\cos\left(\frac{\pi d}{\lambda}\psi\right) \\ \sin\left(\frac{3\pi d}{\lambda}\omega\right)\sin\left(\frac{\pi d}{\lambda}\psi\right) \\ \sin\left(\frac{3\pi d}{\lambda}\omega\right)\sin\left(\frac{3\pi d}{\lambda}\psi\right) \end{bmatrix} \begin{array}{l} \\ \\ \\ \\ \\ \leftarrow\text{remove} \\ \leftarrow\text{remove} \\ \\ \\ \leftarrow\text{remove} \\ \leftarrow\text{remove} \\ \\ \\ \\ \\ \\ \end{array}, \tag{4.8}$$

which yields

$$\tilde{\mathbf{a}}(\theta,\varphi) = 2 \begin{bmatrix} \cos\left(\frac{3\pi d}{\lambda}\omega\right)\cos\left(\frac{3\pi d}{\lambda}\psi\right) \\ \cos\left(\frac{3\pi d}{\lambda}\omega\right)\cos\left(\frac{\pi d}{\lambda}\psi\right) \\ \cos\left(\frac{3\pi d}{\lambda}\omega\right)\sin\left(\frac{\pi d}{\lambda}\psi\right) \\ \cos\left(\frac{3\pi d}{\lambda}\omega\right)\sin\left(\frac{3\pi d}{\lambda}\psi\right) \\ \cos\left(\frac{\pi d}{\lambda}\omega\right)\cos\left(\frac{3\pi d}{\lambda}\psi\right) \\ \cos\left(\frac{\pi d}{\lambda}\omega\right)\sin\left(\frac{3\pi d}{\lambda}\psi\right) \\ \sin\left(\frac{\pi d}{\lambda}\omega\right)\cos\left(\frac{3\pi d}{\lambda}\psi\right) \\ \sin\left(\frac{\pi d}{\lambda}\omega\right)\sin\left(\frac{3\pi d}{\lambda}\psi\right) \\ \sin\left(\frac{3\pi d}{\lambda}\omega\right)\cos\left(\frac{3\pi d}{\lambda}\psi\right) \\ \sin\left(\frac{3\pi d}{\lambda}\omega\right)\cos\left(\frac{\pi d}{\lambda}\psi\right) \\ \sin\left(\frac{3\pi d}{\lambda}\omega\right)\sin\left(\frac{\pi d}{\lambda}\psi\right) \\ \sin\left(\frac{3\pi d}{\lambda}\omega\right)\sin\left(\frac{3\pi d}{\lambda}\psi\right) \end{bmatrix}. \tag{4.9}$$

4.3.3 A Note on the scaling factors

From Equation (4.4) and Equation (4.9), we observe that both $\mathbf{T}$ and $\tilde{\mathbf{a}}(\theta,\varphi)$ have a scaling factor of 1/2 and 2, respectively. The reason for this is due to that the unitary matrix, $\mathbf{T}_{12}$ must fulfill $\mathbf{T}\mathbf{T}^H = \mathbf{I}$. From the definition of $\mathbf{U}$ in Equation (3.36) and $\mathbf{T}$ in Equation (4.4), it can be observed that this would not be fulfilled without the scaling factors. However, observing the search function $\mathbb{P}_{\mathrm{MU}}$ presented in Equation (3.44), we observe that each vector multiplication will be scaled with the same scaling factors. This means that the search function not will be influenced by removing the scaling factors, and this simplification can therefore safely be done.

Chapter 5

Implementation

This chapter is used to describe the implemented design in this thesis. The chapter first defines the goals for the implementation, followed by methods for solving the necessary tasks, and a description of how it is done.

5.1 Task Description

From the presented theory in Section 3.6 and Section 3.7, the algorithm can be divided into five main steps:

1. Sample IQ values using the BLE sampling standard and form the sample-matrix, $\mathbf{X}$.
2. Find the covariance matrix, $\hat{\mathbf{R}}_{xx}$ using the maximum likelihood estimator.
3. Perform the unitary transformation, including FB averaging to obtain the real-valued covariance matrix, $\mathbb{R}_{xx}$.
4. Perform eigenvalue-decomposition (EVD) to obtain the eigenvalues and find signal subspace eigenvector, $\mathbb{E}_s$.
5. Perform the search using $\mathbb{P}_{\mathrm{MU}}$ to find AoA.

From the given steps, we observe that steps 2 to 5 can be beneficial to implement in hardware, as they introduce a lot of repetitive matrix operations. In general, matrix operations are usually parallelizable to some extent, meaning that specialized hardware architectures can be designed to perform better in terms of speed of operation compared to general processing units, where the operations are performed sequentially. We can also observe that the real-valued approach is decided from the arguments given in Section 4.2. The goal is therefore to implement a real-valued MUSIC algorithm on a Field Programmable Gate Array (FPGA). A visualization of how the tasks are divided is shown in Figure 5.1. The system sketch in Figure 2.1 is also updated to the new system architecture, and is it presented in Figure 5.2

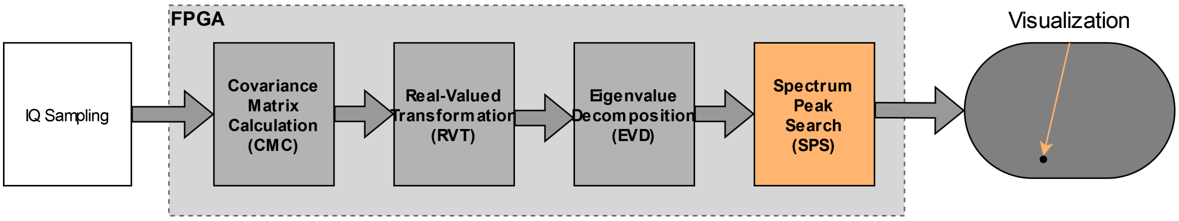

FPGA
IQ Sampling
Covariance Matrix Calculation (CMC)
Real-Valued Transformation (RVT)
Eigenvalue Decomposition (EVD)
Spectrum Peak Search (SPS)
Visualization

Figure 5.1: Block diagram of hardware implementation of the MUSIC algorithm.

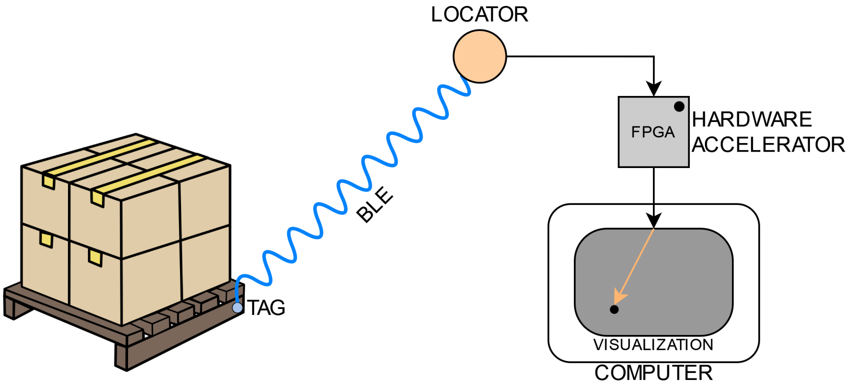


Figure 5.2: Updated system architecture.

This thesis will only focus on the implementation of Spectral Peak Search (SPS), highlighted with orange color in Figure 5.1. Tommy A. Opstad, a fellow student, is implementing the hardware for steps 2 to 4, that is the Covariance Matrix Calculation (CMC), Real-Valued Transformation (RVT), and Eigenvalue Decomposition (EVD) modules presented in Figure 5.1. As the steps 2 to 5 depends on the results from the previous step, we are not able to perform the selected steps in parallel. A summary of his design can be found in Section 5.4 in addition to in his thesis [1]. The desired theoretical maximum error of the MUSIC algorithm is 0.5°.

5.2 Spectral Peak Search Algorithm

As stated in Section 3.6.3, the goal for the MUSIC algorithm is to find the $K = 1$ peaks from the search function in Equation (3.44). It is beneficial to search for the peak using signal subspace, as it requires significantly fewer computations when compared to using noise subspace. Listing 5.1 indicates how a 2D search can be performed in order to achieve this goal.

Algorithm 5.1: Real-valued AoA search loop.

```
input: start_el, stop_el, start_az, stop_az, step_el, step_az, $E_s$
output: peak_el, peak_az
$\theta$ = start_el;
$\varphi$ = start_az
peak_val, peak_el, peak_az = 0,0,0
while $\theta$ < stop_el:
    $\varphi \leftarrow$ start_az
    while $\varphi$ < stop_az:
        curr_val $\leftarrow$ $||\tilde{\mathbf{a}}^H(\theta,\varphi)\mathbb{E}_S||_2$
        if curr_val > peak_val:
            peak_val $\leftarrow$ curr_val
            peak_el $\leftarrow$ $\theta$
            peak_az $\leftarrow$ $\varphi$
        $\varphi \leftarrow \varphi$ + step_az
    $\theta \leftarrow += \theta$ +step_el
end
```

The goal of the algorithm is to find the combination of θ and φ that yields the highest value of $||\tilde{\mathbf{a}}^H(\theta,\varphi)\mathbb{E}_S||_2$. This is a result of deciding to search through the signal subspace, as discussed in Chapter 4. By searching in the signal subspace, we only need to work with vectors, and we observe, from Equation (3.19), that the result from this vector multiplication is a 1×1 value. From the definition of Euclidean norm in Equation (3.20), we observe that the Euclidean norm can be found from the absolute value of the vector multiplication $\tilde{\mathbf{a}}^H(\theta,\varphi)\mathbb{E}_S$.

From Listing 5.1 we observe that the number of iterations and the resolution of the search depend significantly on the step sizes, `step_el` and `step_az`, of the two loops. Lower step sizes yield greater resolution but an increase in number of iterations, and vice versa. Figure 5.3 presents the complexity of the algorithm presented in Listing 5.1, with respect to the step sizes. With complexity, we mean number of loop iterations.

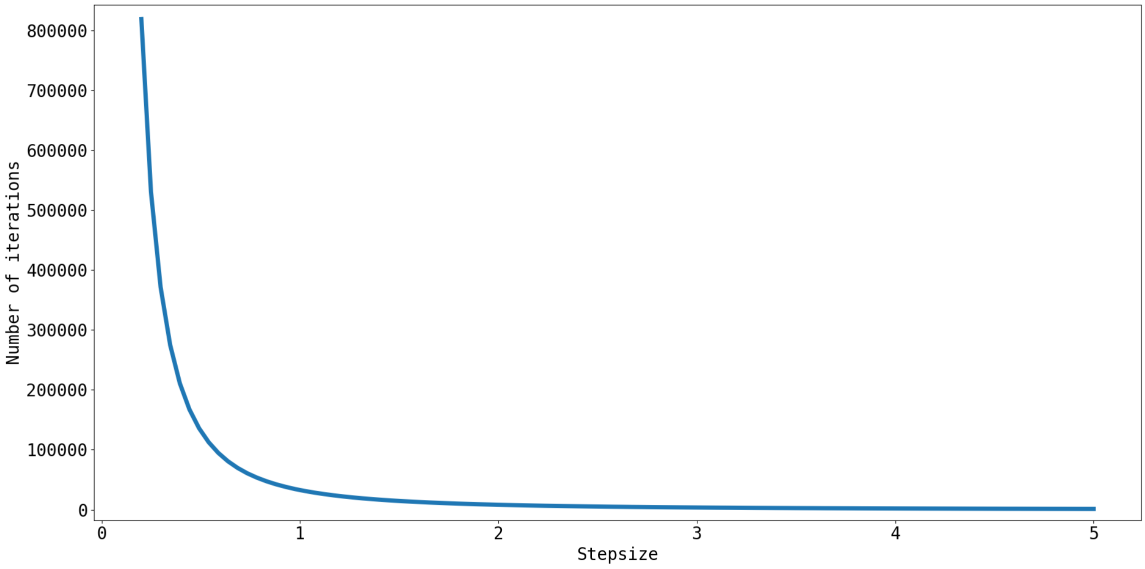


Figure 5.3: Complexity vs. step sizes for the 2D MUSIC search algorithm.

The complexity presented assumes that the search region is $\theta \in [0°, 90°]$ and $\varphi \in [0°, 360°]$. In general, the complexity of Listing 5.1 can be calculated using Equation (5.1).

$$\text{Complexity} = \frac{\texttt{stop\_el} - \texttt{start\_el}}{\texttt{step\_el}} \cdot \frac{\texttt{stop\_az} - \texttt{start\_az}}{\texttt{step\_az}} \tag{5.1}$$

As indicated by Figure 5.3, the search complexity increases exponentially with decreasing step sizes. One way to reduce this trade-off and achieve an acceptable resolution, while keeping the number of iterations relatively low, is by performing multiple searches. The idea with performing multiple searches is that the first search is performed within the complete search region, i.e., $\theta \in [0°, 90°]$ and $\varphi \in [0°, 360°]$. From the first search, we can reduce the region of the next search around the found peak. As the search region is decreased, we can also reduce the step sizes without introducing numerous iterations. This can be repeated until having obtained a search with step sizes resulting in a desired resolution is performed. Figure 5.4 illustrates this idea.

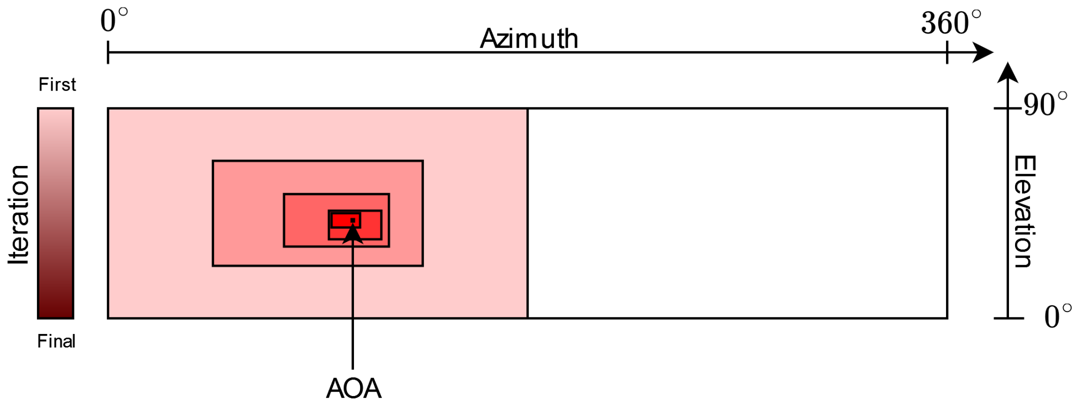


Figure 5.4: Visualization of the theoretical idea with performing multiple search iterations, narrowing the search regions down to find the AoA.

To understand why this can work, the typical spectrum, shown in Figure 5.5, is further used for explaining the method.

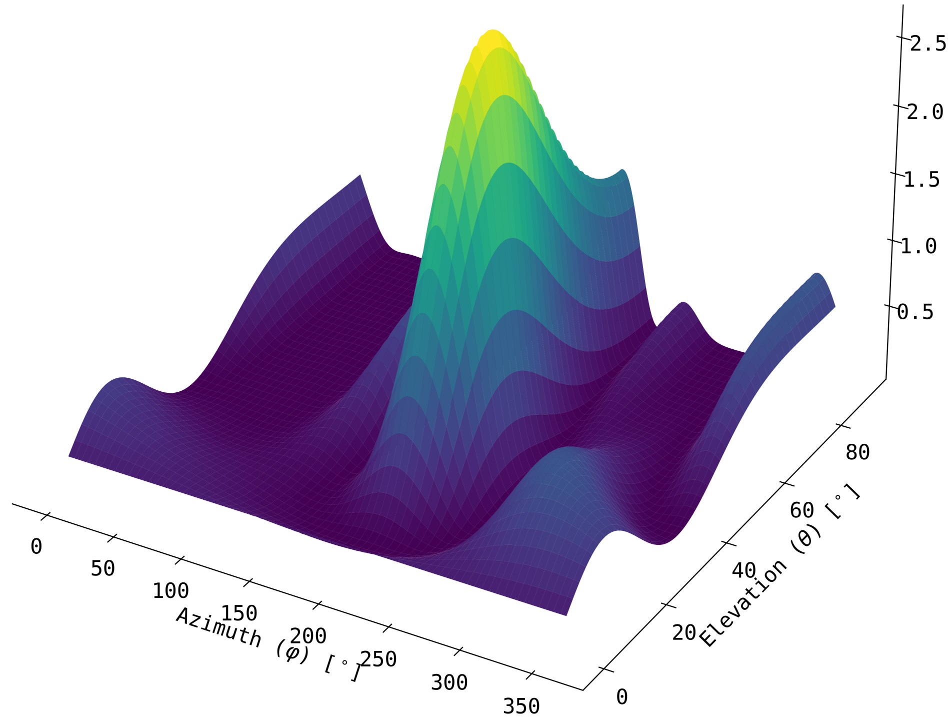


Figure 5.5: RV-MUSIC spectrum for a search region $\theta \in [0°, 90°]$ and $\varphi \in [0°, 360°]$ with SNR=40 dB.

The visualization in Figure 5.5 indicates that there is no abrupt changes in the spectrum, meaning that if appropriate step sizes are chosen, we will be able to localize the approximate peak in the first search iterations. Of course, if the step sizes are too large, it is easy to miss the approximate peak region, and one must be careful when deciding the step sizes. A general equation is formed for calculating the number of iterations needed. Let N denote the number of searches to be performed, and corresponding step sizes, $\boldsymbol{\Delta}\theta = \{\Delta\theta_1, \Delta\theta_2, ..., \Delta\theta_N\}$ and $\boldsymbol{\Delta}\varphi = \{\Delta\varphi_1, \Delta\varphi_2, ..., \Delta\varphi_N\}$.

$$\text{Number of iterations} = \frac{360-0}{\Delta\varphi_1} \cdot \frac{90-0}{\Delta\theta_1} + \sum_{i=2}^{N} \frac{\Delta\varphi_{i-1}}{\Delta\varphi_i} \cdot \frac{\Delta\theta_{i-1}}{\Delta\theta_i} \tag{5.2}$$

The idea behind this equation is that the i^{th}-search is limited around the previous found peak, $p_{i-1} = (\varphi_{p,i-1}, \varphi_{p,i-1})$ by $[\theta_{p,i-1} - \frac{\Delta\theta_{i-1}}{2}, \theta_{p,i-1} + \frac{\Delta\theta_{i-1}}{2}]$ in the elevation region and $[\varphi_{p,i-1} - \frac{\Delta\varphi_{i-1}}{2}, \varphi_{p,i-1} + \frac{\Delta\varphi_{i-1}}{2}]$ in the azimuth region.

The number of iterations for the i^{th} search can therefore be calculated as

$$\begin{aligned} \textbf{Elevation region:} \quad & \frac{\left(\theta_{p,i-1} + \frac{\Delta\theta_{i-1}}{2}\right) - \left(\theta_{p,i-1} - \frac{\Delta\theta_{i-1}}{2}\right)}{\Delta\theta_i} = \frac{\Delta\theta_{i-1}}{\Delta\theta_i} \\ \textbf{Azimuth region:} \quad & \frac{\left(\varphi_{p,i-1} + \frac{\Delta\varphi_{i-1}}{2}\right) - \left(\varphi_{p,i-1} - \frac{\Delta\varphi_{i-1}}{2}\right)}{\Delta\varphi_i} = \frac{\Delta\varphi_{i-1}}{\Delta\varphi_i}. \end{aligned} \tag{5.3}$$

A high-level model is implemented to tests this idea, and a visualization of how the search narrows down to find the simulated AoA is shown in Figure 5.6.

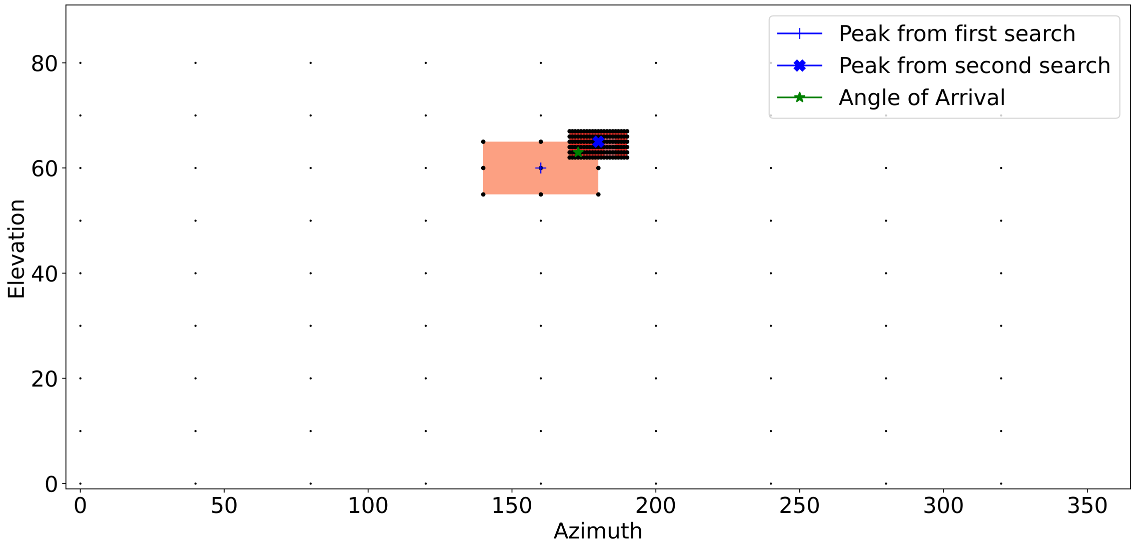


Figure 5.6: A visualization of a simulation in the high level model performed with 3 searches. Each black dot indicates one computation of the search function, $\mathbb{P}_{\text{MU}}$, presented in Equation (3.44). The light red square indicates the second search area, while the dark-red square indicates the third search area.

Some initial tests are performed using the high-level model to indicate the reduction in number of values to compute and compare, observing the time used and the average absolute error for the tests. For each test, a set of 50 simulations is performed with a new, random AoA within the legal region. The number of search iterations is tested from one to six, and the results are presented in Table 5.1.

Table 5.1: High-level simulations of multiple search iterations.

Number of search iterations	Time used [ms]	Loop iterations/complexity	Average absolute error [°]
1	2140	32670	0.08
2	679	8074	0.02
3	298	2109	0.04
4	83	596	0.25
5	31	239	0.58
6	22	143	1.16

The obtained results indicate that using a multiple-search approach can reduce the complexity significantly, but if the initial step sizes are too large, we can experience an increase in the average absolute error.

5.3 Hardware Implementation of Spectral Peak Search

The main benefit of a hardware accelerator is the possibility of parallelism of independent tasks. This section discusses possible solutions for the SPS hardware architecture. In general, the goal is to minimize the time spent on calculating the AoA.

In the following section, multiple figures are used for visualizing theoretical and implemented architectures for the hardware accelerator. For these figures, three symbols are used for describing different dimensions of connected signals between the modules and resources. Table 5.2 presents these symbols with a description and the expected dimensions.

Table 5.2: Architecture communication symbols.

Symbol	Description	Dimensions
⟶	Single bit	1
⇸	Vector	2
⇨	Multiple vectors	> 2

5.3.1 Hardware Resources

For this project, the Zynq-7020 System on Chip (SoC) from Xilinx [16], located on the PYNQ-Z1 development kit, is used as hardware. The implementation of CMC, RVT, and EVD also requires some resources, and an overview of the available resources for the implementation of SPS is presented in Table 5.3.

Table 5.3: Overview of available resources for SPS. The total available amount is found from [16], and the resources used for CMC, RVT, and EVD is found from [1].

	Look-up tables	BRAM	DDR3	DSP slices
Total available	53 200	140	512 MB	220
Used by CMC	3 424	0	0	36
Used by RVT	3 780	6	0	0
Used by EVD	19 218	0	0	0
Free to use for SPS	26 778	134	512 MB	184

5.3.2 Search Core

To perform the search discussed above, certain components are required. Three central functionalities for the search are to be able to multiply the two vectors, $\mathbb{E}_s$ and $\tilde{\mathbf{a}}(\theta, \varphi)$, being able to store the current peak-angle values, and have support for comparing the highest vector multiplication product with the newest vector multiplication. A minimalistic architecture is presented in Figure 5.7.

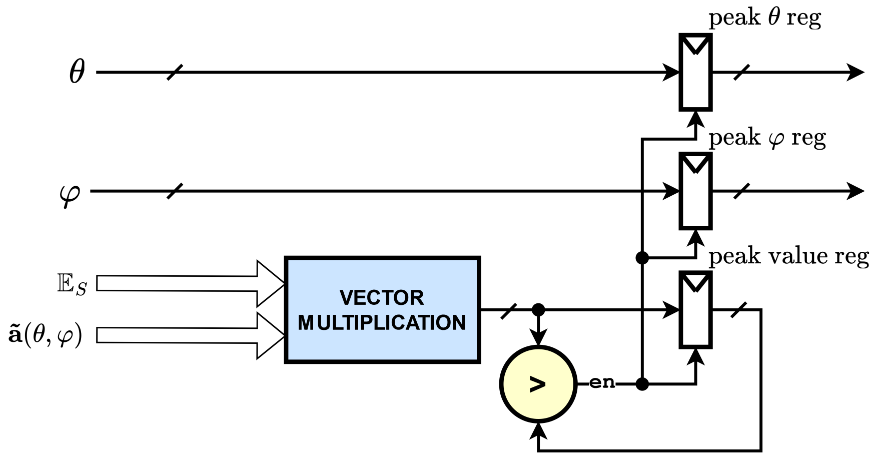


Figure 5.7: A basic architecture for the search to be performed.

The registers holding the peak values are enabled only if the newest vector multiplication results in a higher value than the current peak value. The minimalistic architecture in Figure 5.7 will do the required task. However, this architecture will probably not outperform a software-based approach in terms of speed, as all the search iterations must be performed sequentially. Observing the search loop in Listing 5.1, we notice that all vector multiplications can be done simultaneously, but that we cannot output the actual AoA before all vector multiplications are performed, and the values are compared. Using the architecture in Figure 5.7 will not utilize one of the key elements in a hardware accelerator: parallelism. The architecture presented in Figure 5.7 is therefore modified to support a greater level of parallelism, forming the component named `search core`, with its architecture presented in Figure 5.8.

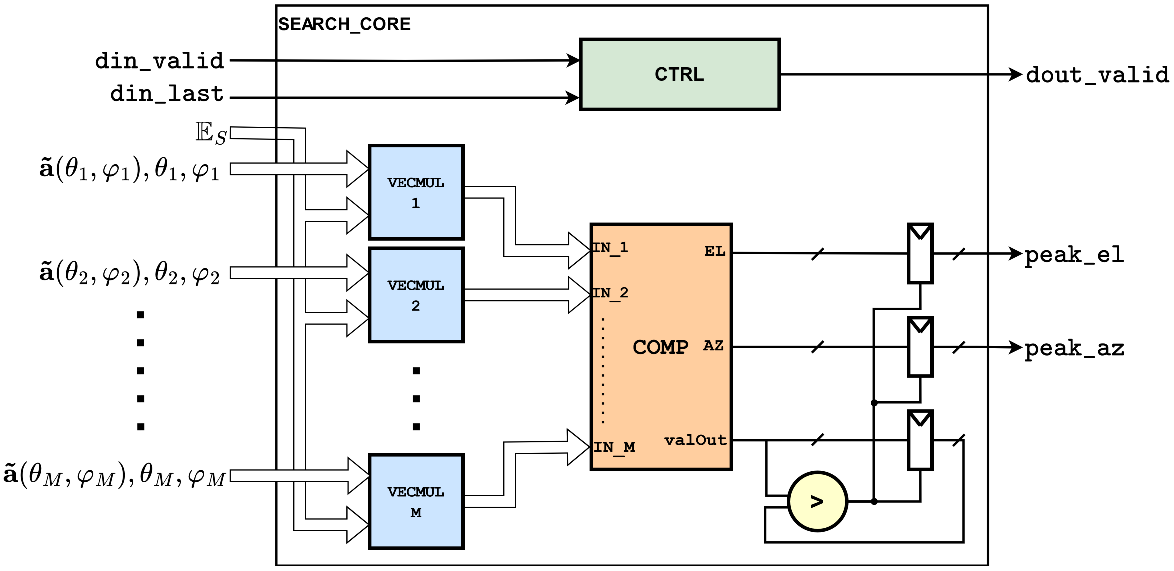

SEARCH_CORE
din_valid
din_last
CTRL
dout_valid
$\mathbb{E}_S$
$\tilde{\mathbf{a}}(\theta_1, \varphi_1), \theta_1, \varphi_1$
$\tilde{\mathbf{a}}(\theta_2, \varphi_2), \theta_2, \varphi_2$
$\tilde{\mathbf{a}}(\theta_M, \varphi_M), \theta_M, \varphi_M$
VECMUL 1
VECMUL 2
VECMUL M
IN_1
IN_2
IN_M
COMP
EL
AZ
valOut
peak_el
peak_az
>

Figure 5.8: Microarchitecture for the Search Core.

The architecture is based on M parallel vector multiplications (`VECMUL`s), with its architecture presented in Section 5.3.3, that feeds the output into a component which compares the M

values and outputs the highest value with corresponding values. The architecture of this component is named `COMP`, and it is further presented in Section 5.3.4. As indicated by Figure 5.8, the angle combinations are fed into the search core together with the corresponding steering vector, and they will follow the calculated values throughout the circuit. Another approach could be to assign an ID for each steering-vector value, and then add logic for reverse-calculating the correct angle combinations when the search is complete. The chosen approach will likely require more registers for storing the angle combinations throughout the circuit compared to the other presented approach, as it requires two values to be stored compared to one throughout the circuit. However, it will not require more logic and resources for calculating the angle combinations after the search has been completed. This will also yield an immediate AoA-result when the peak values are found without adding extra clock cycles to the search.

Ideally, one would want M to represent the maximum possible combinations of θ and φ, such that all values of $\mathbb{P}_{\text{MU}}$ in Equation (3.44) could be computed in parallel. However, this is not achievable with the available hardware resources presented in Table 5.3. This means that the search core must be able to receive the steering vectors and corresponding angles over multiple clock cycles. This implies that the output of `COMP` will only be the highest `VECMUL` result with corresponding angles out of the M values that are fed into the search core. To find the global maximum, one would need to store the global peak value, and compare this to the output of `COMP`. The result of this comparison enables the peak registers shown in Figure 5.8.

To maximize throughput, we want the components of the search core to support pipelining such that we can feed M steering vectors each clock cycle. This is achieved by adding pipelining registers inside both `VECMUL` and `COMP`.

Behavior of the Search Core

As presented in the above section, the search core must be able to receive the steering vector values over multiple clock cycles as we are unable to achieve an architecture that supports fully parallel computations. The total number of values to input could be a fixed number, allowing a counter to indicate when to end the search. However, as we would like to reuse the search core for multiple search iterations, as presented in Section 5.2, this method is not preferred. Two control signals are therefore added as inputs. `dataIn_valid` indicates that the input is valid, and when this is set high, the search is to be performed. To indicate that the inputs no longer will be valid, and that the search is to be completed, `dataIn_last` is set high for one clock cycle together with the M last steering vector values. This allows us to feed the input data for as long as one would like, resulting in a highly flexible behavior. This behavior is presented in the timing diagram in Figure 5.9, and it means that when the final output of `COMP` is compared to the global peak value, `dataOut_valid` can be set high, indicating that the output is valid.

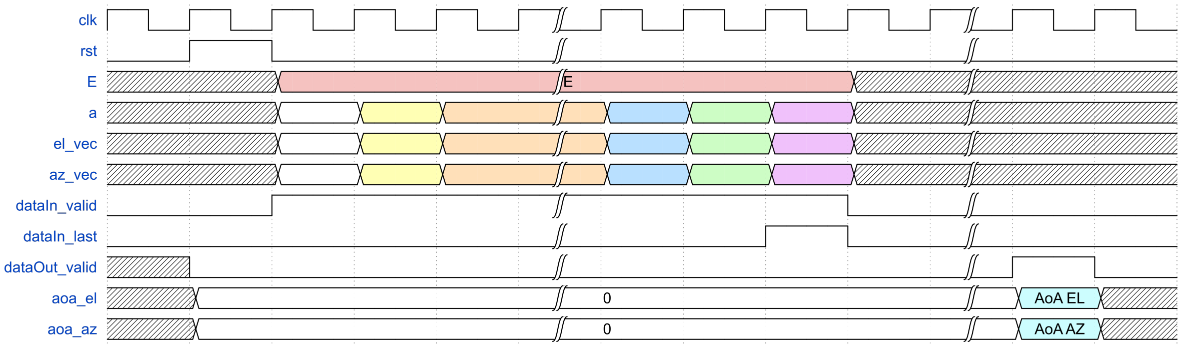


Figure 5.9: Example timing diagram for the search core.

As indicated by Figure 5.9, there are four inputs to the search core that contains data, `E`, `a`, `el_vec`, and `az_vec`. The input `E` consists of $D = 12$ signed values with N bits per value. Input `a` consists of M steering vector values, all consisting of $D = 12$ signed values of N bits, while `el_vec` and `az_vec` consists of M unsigned values of 12 bits. The latter two inputs can both be unsigned as they only need to hold positive values due to that the search regions are $\theta \in [0°, 90°]$ and $\varphi \in [0°, 360°]$. The m^{th} $(1 \leq m \leq M)$ steering vector and angle values are mapped directly to the m^{th} instantiation of the `VECMUL` module in the search core. This can also be observed in Figure 5.8.

5.3.3 Vector Multiplication Unit

As mentioned in the previous section, one central functionality for the search is to perform the vector multiplication between the steering vector and signal subspace eigenvector. The following section is used to present the architecture for the vector multiplication unit.

From Equation (3.19), we observe that each of the values e_i and a_i can be multiplied in parallel before the sum of the parallel multiplication must be found. Of course, the multiplication can also be done sequentially, with a Multiply-Accumulate (MAC) architecture as shown in Figure 5.10.

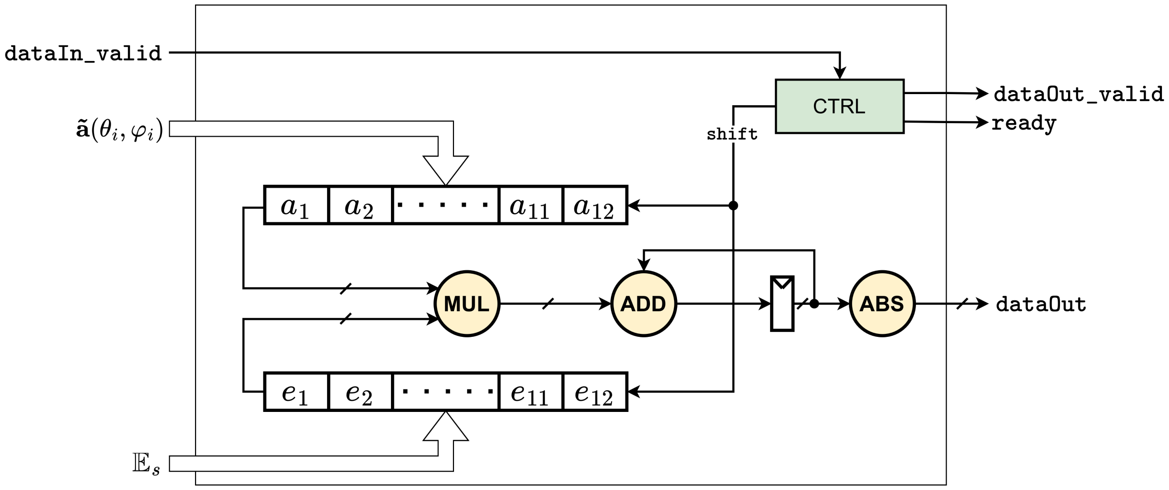


Figure 5.10: MAC architecture for the `VECMUL` unit.

The MAC architecture for one `VECMUL` unit will in theory only require one DSP slice to perform the multiplication. The architecture is based on shift registers for storing- and outputting the correct values for $\tilde{\mathbf{a}}(\theta, \varphi)$ and $\mathbb{E}_s$ to the multiplier, as indicated in Figure 5.10. When `shift` is set high, the values in the vectors are shifted once to the left, and new values will be available at the output. This approach will not be able to receive new steering vector values each clock cycle, as it will require $D = 12$ clock cycles to shift all vector values into the multiplier. A `ready` signal must therefore be added to ensure that the vectors are not overwritten while the operation is ongoing. The `ready` signal is set high to indicate that it is ready for new values simultaneously as `dataOut_valid` is set high, indicating that the multiplication is complete and that `dataOut` is valid. The total number of cycles needed to perform the vector multiplication with this architecture is at least 12

Another architecture is also made for supporting the idea of being able to input one new steering vector value each clock cycle. The architecture presented in Figure 5.11 performs the $D = 12$ multiplications in parallel, before a parallel approach for the addition is performed. Pipelining registers are added to support the idea of inputting new values each clock cycle. By shifting the angle values throughout the `VECMUL` unit as indicated in Figure 5.11, no extra control logic is needed to ensure that the correct angle values are available at the output with corresponding `dataOut`.

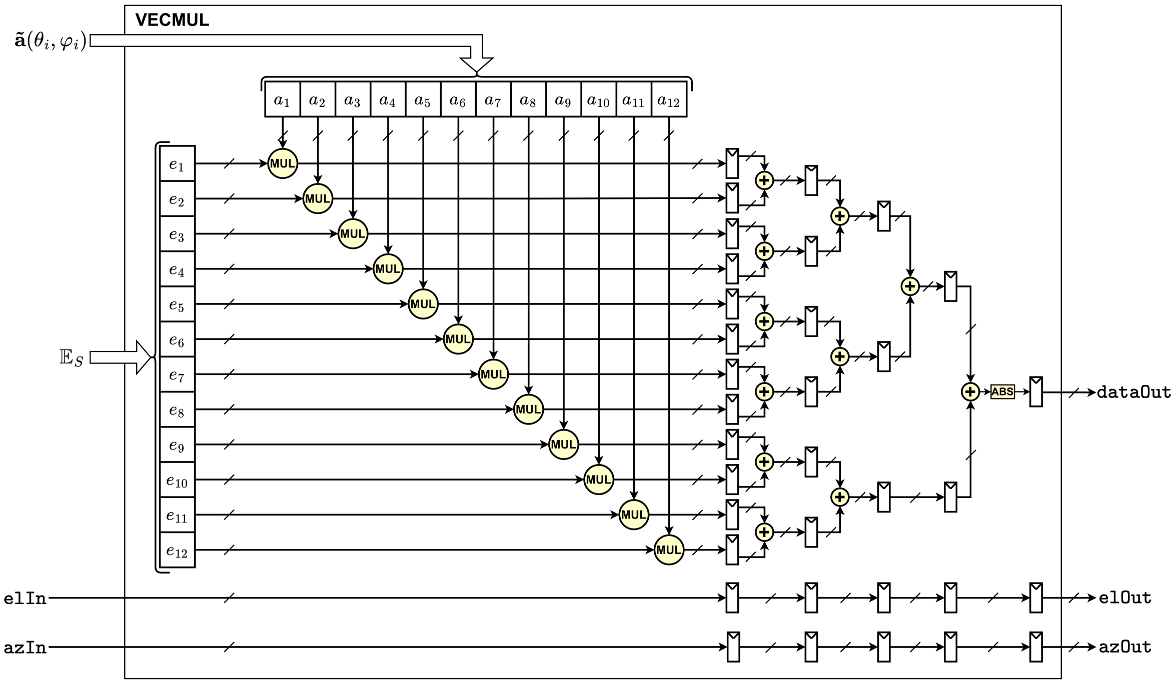

VECMUL
$\tilde{\mathbf{a}}(\theta_i, \varphi_i)$
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12}
$\mathbb{E}_S$
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9 e_{10} e_{11} e_{12}
MUL
ABS
dataOut
elIn
elOut
azIn
azOut

Figure 5.11: Microarchitecture of the `VECMUL` module.

Compared to the architecture in Figure 5.10, the parallel architecture will require more DSP slices, which reduces the maximum value of M. The main limiting resource for this architecture is the DSP slices. From Table 6.4, we observe that each `VECMUL` unit is assigned 16 DSP slices per instantiation during synthesis. From Table 5.3, we know that we have 184 DSP slices available for the SPS, limiting $M_{\text{max}} = 184/16 = 11$ parallel `VECMUL` units.

The latter architecture is chosen for this accelerator, as it allows us to fully pipeline the values into the search core. Another factor for the decision is presented in Section 5.3.5, and to summarize, the way the steering vector values is fed into the search core will also limit $M_{\max} = 11$. It would therefore not be beneficial to use the MAC architecture for the `VECMUL` unit, as we would be unable to feed enough steering vector values to the search core.

5.3.4 Comparison Unit

From Figure 5.8, we observe that the M `VECMUL`-outputs and corresponding angles are fed into the comparison unit (`COMP`). The goal for this unit is to output the highest value and the corresponding angles. This can effectively be achieved with the parallel comparison tree architecture presented in Figure 5.12. The figure holds the architecture for $M = 10$, and the architecture changes with M.

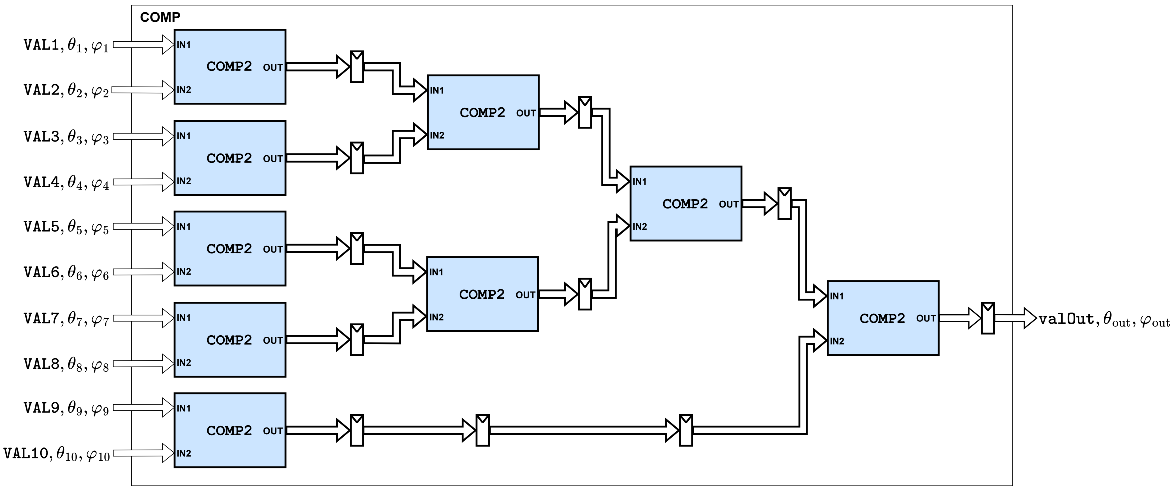

COMP
VAL1, θ_1, φ_1
VAL2, θ_2, φ_2
VAL3, θ_3, φ_3
VAL4, θ_4, φ_4
VAL5, θ_5, φ_5
VAL6, θ_6, φ_6
VAL7, θ_7, φ_7
VAL8, θ_8, φ_8
VAL9, θ_9, φ_9
VAL10, $\theta_{10}, \varphi_{10}$
IN1
IN2
COMP2
OUT
valOut, $\theta_{\text{out}}, \varphi_{\text{out}}$

Figure 5.12: Microarchitecture for the `COMP` unit with $M = 10$.

The architecture supports pipelining, which means that new values can be loaded into `COMP` each clock cycle. The `COMP` unit is build by using `COMP2` units. The architecture for the `COMP2` unit is presented in Figure 5.13, and it takes two values with corresponding angles and outputs the largest value with corresponding angles. By comparing the `COMP2` outputs as shown in Figure 5.12, we can find the maximum value and corresponding values after 5 clock cycles.

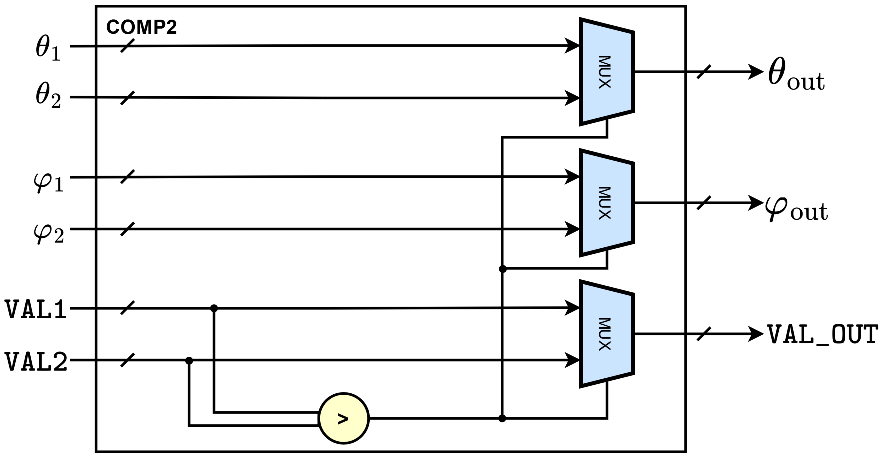


Figure 5.13: Architecture for the `COMP2` unit.

The output of the `COMP2` entity is controlled by the result from `VAL1` > `VAL2`. The truth table for `COMP2` is shown in Table 5.4. The truth table also indicates that in the event of `VAL1` and `VAL2` being equal, `VAL2` and the corresponding angles are outputted. This is ok, due to that without any knowledge of the expected output, we cannot decide if either of the inputs is more correct in any other way than evaluating the search function. If the vector multiplication of two different steering vectors yields the same value, then they are both equally correct. From the discussion of how the steering vectors are fed into the search core, which is presented in Section 5.3.5, we know that the steering vector values fed into the search core will be very similar, and in some cases equal due to the values being quantized. Equal steering vectors will yield equal values to be compared, and one must be aware of this situation.

Table 5.4: Truth table for the COMP2 entity.

`VAL1` > `VAL2`	`VAL_OUT`	θ_{out}	φ_{out}
True	`VAL1`	θ_1	φ_1
False	`VAL2`	θ_2	φ_2

From the synthesis reports of `VECMUL` and `COMP`, Section 6.2, we observe that `COMP` outperforms the `VECMUL` unit in terms of maximum frequency. The `COMP` unit consists of simple arithmetic operations, and the `VECMUL` unit includes multiplication. The architecture for `COMP` in Figure 5.12 could therefore be modified to perform larger comparisons per clock cycle, which again would form a longer critical path, hence lowering the maximum frequency. This could be beneficial, as it would require fewer clock cycles for one full cycle of the comparison to complete. However, since the architecture will be fully pipelined, this modification would not influence the total number of clock cycles significantly. Using the presented architecture for `COMP` ensures that this part of the design will not be the bottleneck of the total design, presented in Section 5.4.

5.3.5 Obtaining the Steering Vectors

Another key factor for minimizing the time spent for the SPS is to have an efficient way of obtaining the needed steering vectors, $\tilde{\mathbf{a}}(\theta, \varphi)$. One way of achieving this could be done by implementing hardware support for trigonometric operations such as sine and cosine computations, as well as multiplication for the trigonometric arguments shown in Equation (4.9). This approach would work, but it would require DSP slices and additional hardware resources, most likely limiting the level of parallelism for the search core presented in Section 5.3.2.

Another way of obtaining the steering vector values could be from memory. From Equation (4.9), we observe that all values of $\tilde{\mathbf{a}}(\theta, \varphi)$ can be precomputed and stored in memory. This is because the steering vector only contains information on the antenna array for given variations of θ and φ, and that the array used in this project is known and will not change during run time. The antenna structure is presented in Figure 4.1.

From [14], we observe that there are two types of memory available on the PYNQ Z1 board, DDR3 and BRAM. The DDR3 memory is not integrated on the FPGA and requires external interfacing, such as Advanced eXtensible Interface (AXI), for communication [14]. However, the DDR3 memory offers more available memory compared to the BRAM which is integrated in the FPGA. It is therefore a trade-off between required memory and system complexity. With agreement from EmLogic, it was decided to aim for using the available BRAM, as this would result in a more flexible system, not requiring other resources than the FPGA itself. The following subsection will therefore only discuss how the BRAM is used for storing- and reading the steering vector values.

BRAM structure

From Table 5.3, we observe that we have 134 BRAMs of 36 Kb available for the SPS. From Section 2.4.1, we remember that there is a limited number of possible configurations for the BRAM. Reading one value from a single BRAM takes one clock cycle [17, p. 17], meaning that $D = 12$ BRAMs could be structured in parallel, holding one element each for a given $\tilde{\mathbf{a}}(\theta, \varphi)$, such that one full steering vector could be read every clock cycle. These 12 parallel BRAMs are grouped together, forming a BRAM Group (BG). The structure for a BG is shown in Figure 5.14.

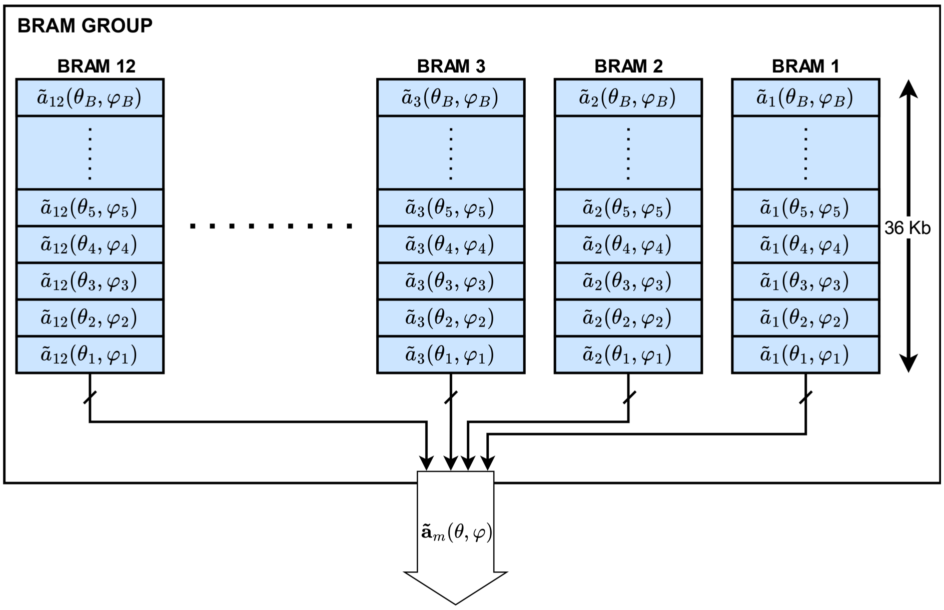

BRAM GROUP
BRAM 12
BRAM 3
BRAM 2
BRAM 1
$\tilde{a}_{12}(\theta_B,\varphi_B)$
$\tilde{a}_{12}(\theta_5,\varphi_5)$
$\tilde{a}_{12}(\theta_4,\varphi_4)$
$\tilde{a}_{12}(\theta_3,\varphi_3)$
$\tilde{a}_{12}(\theta_2,\varphi_2)$
$\tilde{a}_{12}(\theta_1,\varphi_1)$
$\tilde{a}_{3}(\theta_B,\varphi_B)$
$\tilde{a}_{3}(\theta_5,\varphi_5)$
$\tilde{a}_{3}(\theta_4,\varphi_4)$
$\tilde{a}_{3}(\theta_3,\varphi_3)$
$\tilde{a}_{3}(\theta_2,\varphi_2)$
$\tilde{a}_{3}(\theta_1,\varphi_1)$
$\tilde{a}_{2}(\theta_B,\varphi_B)$
$\tilde{a}_{2}(\theta_5,\varphi_5)$
$\tilde{a}_{2}(\theta_4,\varphi_4)$
$\tilde{a}_{2}(\theta_3,\varphi_3)$
$\tilde{a}_{2}(\theta_2,\varphi_2)$
$\tilde{a}_{2}(\theta_1,\varphi_1)$
$\tilde{a}_{1}(\theta_B,\varphi_B)$
$\tilde{a}_{1}(\theta_5,\varphi_5)$
$\tilde{a}_{1}(\theta_4,\varphi_4)$
$\tilde{a}_{1}(\theta_3,\varphi_3)$
$\tilde{a}_{1}(\theta_2,\varphi_2)$
$\tilde{a}_{1}(\theta_1,\varphi_1)$
36 Kb
$\tilde{\mathbf{a}}_m(\theta,\varphi)$

Figure 5.14: Structure of one BRAM group. $\tilde{\mathbf{a}}_m(\theta,\varphi)$ indicates that this is the m^{th} BRAM group where $m \in [1, M]$.

This is more efficient in terms of speed compared to storing all 12 values for the same steering vector in the same BRAM, as this solution would require 12 clock cycles to read the same data. However, the proposed BRAM structure will potentially use more BRAM unless we are able to use the whole depth of each BRAM, i.e., 36 Kb.

The number of steering vector values each BRAM can hold, B, depends on the numbers of bits used per value. A complete overview of the available BRAM configurations is presented in Table 2.2. The more bits used per value, the fewer values can be stored in a single BRAM of 36 Kb. The number of bits used per value also influences the precision of each value in terms of decimal precision, which again can influence the precision of the search. With the design goal of a maximum theoretical error for the search set at 0.5°, we know that we must be able to store all values with 1° step size, i.e., $\tilde{\mathbf{a}}(\theta,\varphi)$ for $\theta \in [0°, 1°, ..., 89°, 90°]$ and $\varphi \in [0°, 1°, ..., 359°, 360°]$. This is a total of $91 \cdot 361 = 32851$ steering vectors. Figure 5.15 visualizes how the number of bits per steering vector value influences the number of full BRAMs required. This result is obtained using the possible configurations in Table 2.2 and assuming that all BRAMs are fully utilized.

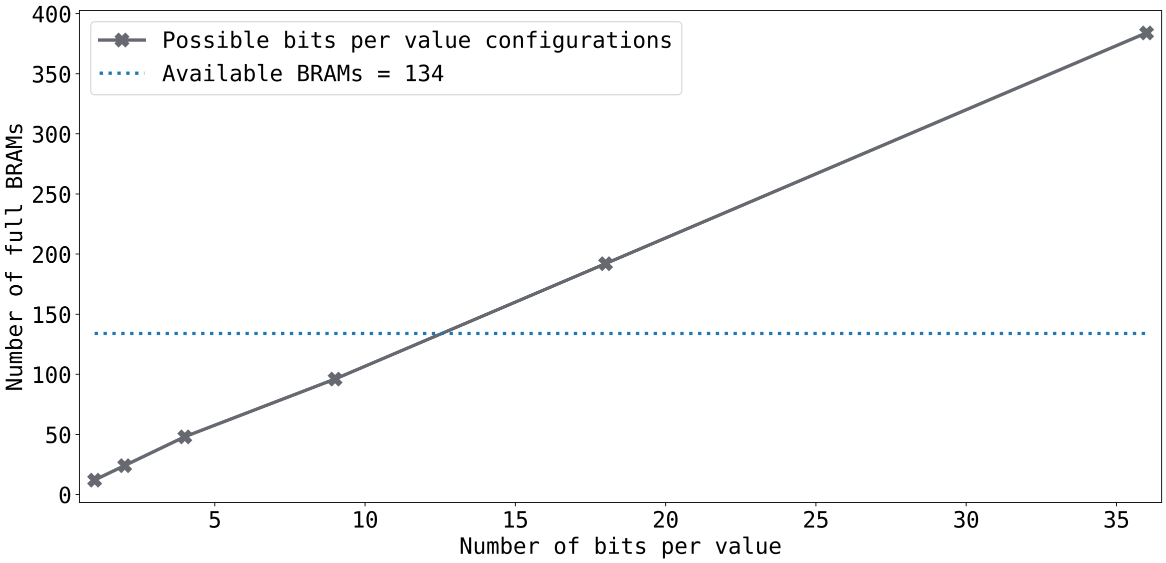


Figure 5.15: A plot visualizing how the number of bits used per steering vector value influences the needed BRAMs if they are stored with 1° step size.

Figure 5.15 indicates that we are not able to store the desired steering vector values with more than 8 bits per value with the available BRAM. Another important factor to be considered is how each value should be stored in the BRAM. We know that the values of the steering vector will be in the range $[-1, 1]$, as they all are products of sine and cosine values. There are three well known solutions for storing real numbers in hardware. The first two are fixed- and floating point numbers. The theory for such value representation are explained in Section 3.2.2 and Section 3.2.1.

While both fixed- and floating point representation of numbers are two well known, and adapted methods, they still often introduce some rounding error. This is often due to a limiting number of bits available. Initial tests are therefore performed on the high level model to observe how the number of decimal places influences the precision of the search. The high-level model of the search, implemented in Python, is modified to round of all steering vector values and signal subspace values using the built-in function `round(a, n)` [37], where the value `a` is rounded to `n`-digits precision. The results are presented in Figure 5.16.

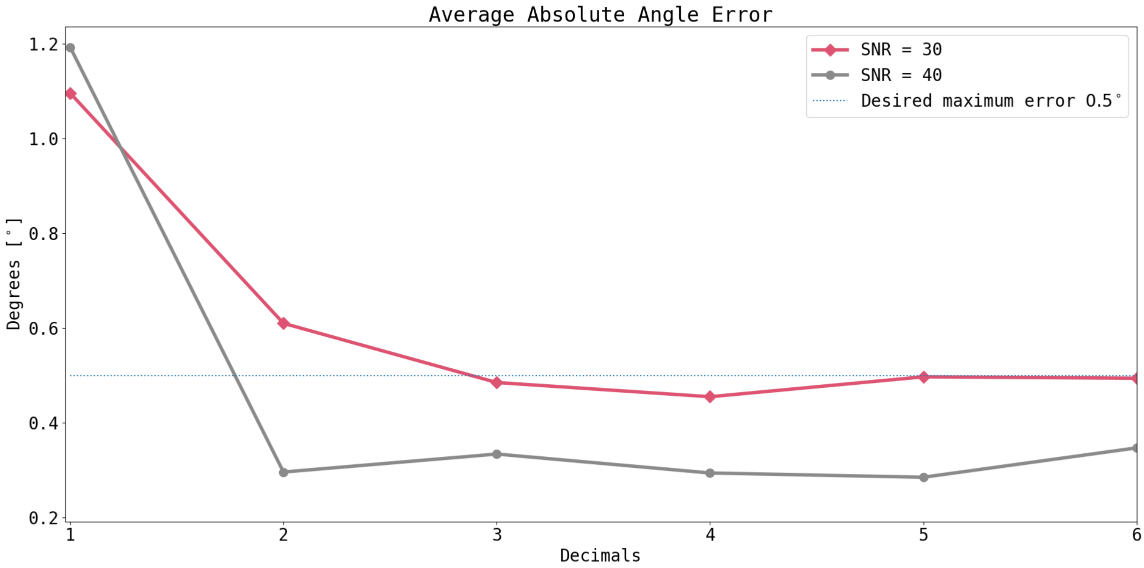


Figure 5.16: High-level model simulation for different number of decimals used. A series of 50 simulations with random AoA is performed per number of decimal test.

From the performed tests, we observe that we do not really need high precision in terms of decimals to obtain the desired precision for the search. From these results, another solution for storing the values in BRAM became feasible. Due to the relatively low required precision needed, it will be possible to scale all steering vector values with a value 10^n, where $n \geq 1$ is the required number of decimals, and round the number to a signed integer value. We observe that by scaling the steering vector values with the value 10^n will result in a possible range of $[-10^n, 10^n]$, which is the same as n digits precision. This method of implementation is preferred, as it allows the architecture to use only signed numbers, which is easier to work with compared to fixed- and floating point numbers. This method is therefore chosen. Table 5.5 holds the number of bits required for representing the scaled values of n-decimal precision together with the integer range.

Table 5.5: Number of bits needed per value for different precision.

n	Integer Range	Number of Bits Needed
1	[-10, 10]	5
2	[-100, 100]	8
3	[-1000, 1000]	10
4	[-10000, 10000]	14
5	[-100000, 100000]	17

From Figure 5.16, we observe that we should have $n \geq 3$ as our required decimal precision. Observing Table 5.5 and Figure 5.15, we see that with the available BRAM, this is not feasible as it would require 10 bits per value, and Figure 5.15 indicates that 8 bits are the maximum values of bits possible. A closer evaluation of the steering vector is therefore performed, where the goal is to observe if we can exploit some symmetrical properties of the steering vector. With the knowledge of the steering vector containing trigonometric values only from Equation (4.9), the possibility of symmetric properties could potentially allow us to reuse the same steering vector values for symmetrical angle values. For the given search region, only the legal values for φ contain symmetric angles. Instead of defining the region for $\varphi \in [0°, 360°]$, we divide it into two regions, $\varphi_n \in [-180°, 0°)$ and $\varphi_p \in [0°, 180°)$. Observing the unit circle, we know that the two definitions, $[\varphi]$ and $[\varphi_n, \varphi_p]$, describe the same regions. We also know that a value from the region $\varphi_h \in [180°, 360°]$ can be defined using a value from φ_n, with the relation

$$\varphi_h = 360° + \varphi_n. \tag{5.4}$$

E.g., we know that the angle $\varphi = -90°$ is the same as

$$\varphi_h = 360° + (-90°) = 270°.$$

The useful observation is done while comparing the steering vector values in the two regions, i.e observing $\tilde{\mathbf{a}}(\theta, -\varphi)$ and $\tilde{\mathbf{a}}(\theta, \varphi)$. To fully understand the observation, one must remember the trigonometric properties

$$\begin{aligned} \cos(-x) &= \cos(x) \\ \sin(-x) &= -\sin(x), \end{aligned} \tag{5.5}$$

which again yield

$$\begin{aligned} \omega(\theta, -\varphi) &= \sin(\theta)\cos(-\varphi) = \sin(\theta)\cos(\varphi) = \omega(\theta, \varphi) \\ \psi(\theta, -\varphi) &= \sin(\theta)\sin(-\varphi) = -\sin(\theta)\sin(\varphi) = -\psi(\theta, \varphi). \end{aligned} \tag{5.6}$$

By inserting these observations into Equation (4.9), we observe some useful properties. They are presented in Equation (5.7).

$$
\begin{aligned}
\tilde{\mathbf{a}}(\theta, -\varphi) &= \begin{bmatrix}
\cos\left(\frac{3\pi d}{\lambda}\omega\right)\cos\left(\frac{3\pi d}{\lambda} - \psi\right)\\
\cos\left(\frac{3\pi d}{\lambda}\omega\right)\cos\left(\frac{\pi d}{\lambda} - \psi\right)\\
\cos\left(\frac{3\pi d}{\lambda}\omega\right)\sin\left(\frac{\pi d}{\lambda} - \psi\right)\\
\cos\left(\frac{3\pi d}{\lambda}\omega\right)\sin\left(\frac{3\pi d}{\lambda} - \psi\right)\\
\cos\left(\frac{\pi d}{\lambda}\omega\right)\cos\left(\frac{3\pi d}{\lambda} - \psi\right)\\
\cos\left(\frac{\pi d}{\lambda}\omega\right)\sin\left(\frac{3\pi d}{\lambda} - \psi\right)\\
\sin\left(\frac{\pi d}{\lambda}\omega\right)\cos\left(\frac{3\pi d}{\lambda} - \psi\right)\\
\sin\left(\frac{\pi d}{\lambda}\omega\right)\sin\left(\frac{3\pi d}{\lambda} - \psi\right)\\
\sin\left(\frac{3\pi d}{\lambda}\omega\right)\cos\left(\frac{3\pi d}{\lambda} - \psi\right)\\
\sin\left(\frac{3\pi d}{\lambda}\omega\right)\cos\left(\frac{\pi d}{\lambda} - \psi\right)\\
\sin\left(\frac{3\pi d}{\lambda}\omega\right)\sin\left(\frac{\pi d}{\lambda} - \psi\right)\\
\sin\left(\frac{3\pi d}{\lambda}\omega\right)\sin\left(\frac{3\pi d}{\lambda} - \psi\right)
\end{bmatrix}\\
&= \begin{array}{cc}
\begin{bmatrix}
\cos\left(\frac{3\pi d}{\lambda}\omega\right)\cos\left(\frac{3\pi d}{\lambda}\psi\right)\\
\cos\left(\frac{3\pi d}{\lambda}\omega\right)\cos\left(\frac{\pi d}{\lambda}\psi\right)\\
-\cos\left(\frac{3\pi d}{\lambda}\omega\right)\sin\left(\frac{\pi d}{\lambda}\psi\right)\\
-\cos\left(\frac{3\pi d}{\lambda}\omega\right)\sin\left(\frac{3\pi d}{\lambda}\psi\right)\\
\cos\left(\frac{\pi d}{\lambda}\omega\right)\cos\left(\frac{3\pi d}{\lambda}\psi\right)\\
-\cos\left(\frac{\pi d}{\lambda}\omega\right)\sin\left(\frac{3\pi d}{\lambda}\psi\right)\\
\sin\left(\frac{\pi d}{\lambda}\omega\right)\cos\left(\frac{3\pi d}{\lambda}\psi\right)\\
-\sin\left(\frac{\pi d}{\lambda}\omega\right)\sin\left(\frac{3\pi d}{\lambda}\psi\right)\\
\sin\left(\frac{3\pi d}{\lambda}\omega\right)\cos\left(\frac{3\pi d}{\lambda}\psi\right)\\
\sin\left(\frac{3\pi d}{\lambda}\omega\right)\cos\left(\frac{\pi d}{\lambda}\psi\right)\\
-\sin\left(\frac{3\pi d}{\lambda}\omega\right)\sin\left(\frac{\pi d}{\lambda}\psi\right)\\
-\sin\left(\frac{3\pi d}{\lambda}\omega\right)\sin\left(\frac{3\pi d}{\lambda}\psi\right)
\end{bmatrix}
&
\begin{matrix}
0\\ 1\\ 2\\ 3\\ 4\\ 5\\ 6\\ 7\\ 8\\ 9\\ 10\\ 11
\end{matrix}
\end{array}
\end{aligned}
\tag{5.7}
$$

From Equation (5.7), we observe that $\tilde{\mathbf{a}}(\theta, \varphi) = \tilde{\mathbf{a}}(\theta, -\varphi)$ when taking the negative values of the steering vector values with index 2, 3, 5, 7, 10, and 11 in $\tilde{\mathbf{a}}(\theta, -\varphi)$. This means that we only need to store the half of the originally estimated steering vector values, as we can use the angle combinations $\theta \in [0°, 90°]$ and $\varphi \in [0°, 180°]$ to genreate steering vector values for the whole region. This is achieved by taking the two's complement of the values with index 2, 3, 5, 7, 10, and 11 in every steering vector for $\varphi \in \varphi_h$. A VHDL function is created to do this task, and is presented in Listing 5.2.

Listing 5.2: VHDL function for modifying the steering vector values for $180 < \varphi \leq 360$.

```vhdl
function modify_steering(sVec : steering_vector) return steering_vector is
    variable retVec : steering_vector;
begin
    for i in 0 to VECTOR_DIM - 1 loop
        if (i = 2 or i = 3 or i = 5 or i = 7 or i = 10 or i = 11) then
            retVec(i) := - sVec(i);
        else
            retVec(i) := sVec(i);
        end if;
    end loop;
    return retVec;
end function; -- modify_steering
```

With this discovery, we observe from Figure 5.17 that we now can use the 2Ki×18 bit configuration of each BRAM, which is presented in Table 2.2. From Table 5.5 and Figure 5.16, we observe that using 18 bits per value yields $n = 5$ decimal precision, and from the initial high level simulations, presented in Figure 5.16, this seems to be sufficient for achieving the desired maximum error of 0.5°.

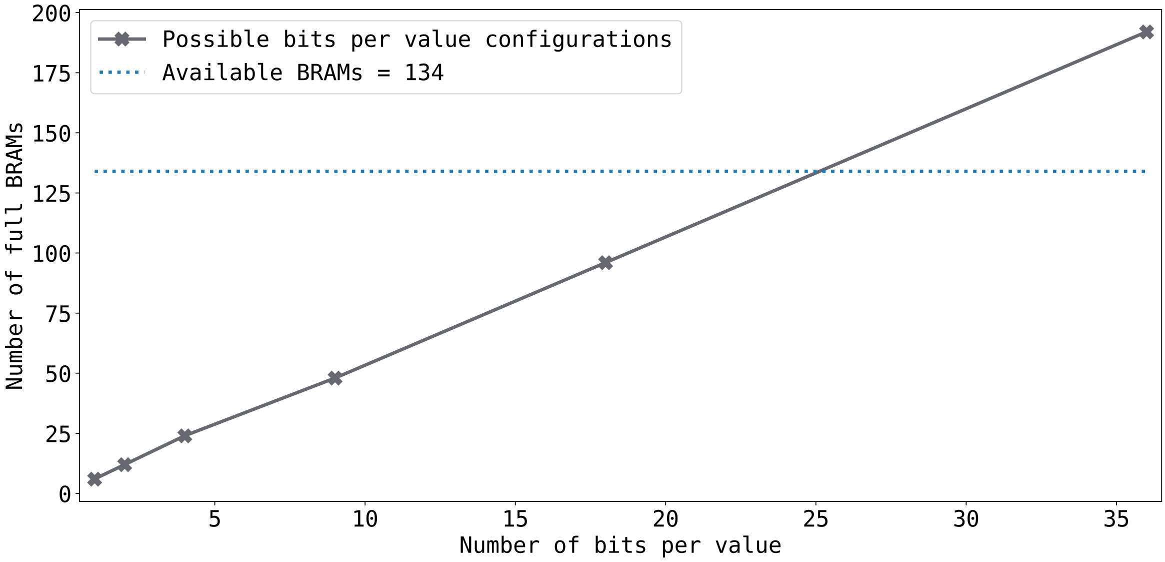


Figure 5.17: A plot visualizing how the number of bits used per steering vector value influences the needed BRAMs if they are stored with 1° step size, using the symmetrical property of the steering vector.

To support the parallel architecture in Figure 5.8, M BGs should be structured in parallel, allowing us to read M full steering vectors each clock cycle [17]. From Table 5.3, it can be observed that we can fit a maximum of $M_{\text{max}} = \text{floor}(134/12) = 11$ BGs in parallel. The final BRAM structure is presented in Figure 5.18, and Table 5.6 presents an extract of how the data is stored. A complete table with the complete BRAM structure can be found in the git repository [1].

[1] https://github.com/EmLogic-Students/HDL/tree/main/sources/SPS/scripts/BRAM

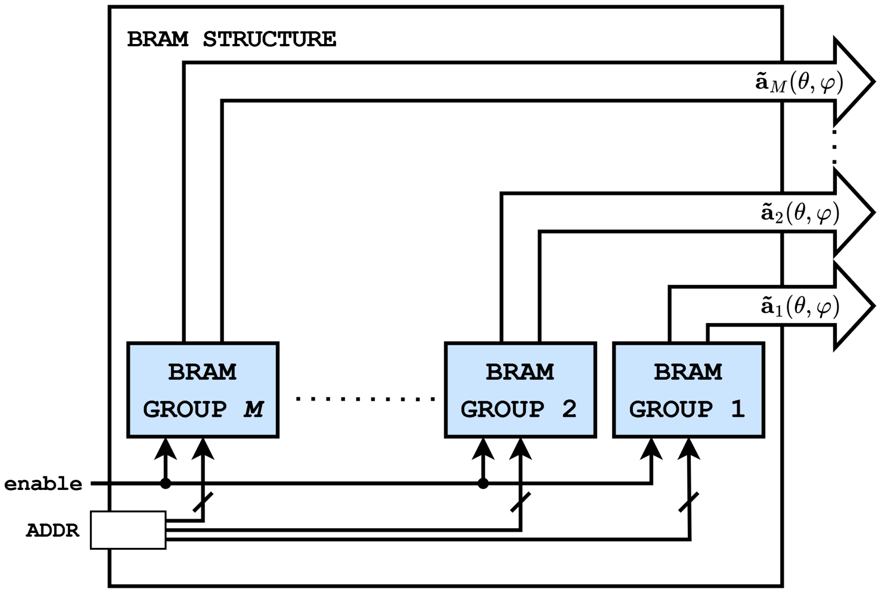

BRAM STRUCTURE
$\tilde{\mathbf{a}}_M(\theta, \varphi)$
$\tilde{\mathbf{a}}_2(\theta, \varphi)$
$\tilde{\mathbf{a}}_1(\theta, \varphi)$
BRAM GROUP *M*
BRAM GROUP 2
BRAM GROUP 1
enable
ADDR

Figure 5.18: The proposed BRAM structure.

Table 5.6: An extract of the final BRAM structure for the Spectral Peak Search.

Common Elevation	BG 1 Azimuth	BG 2 Azimuth	BG 3 Azimuth	BG 4 Azimuth	BG 5 Azimuth	BG 6 Azimuth	BG 7 Azimuth	BG 8 Azimuth	BG 9 Azimuth	BG 10 Azimuth	BRAM index	Abbreviations
0	0 / 360	1 / 359	2 / 358	3 / 357	4 / 356	5 / 355	6 / 354	7 / 353	8 / 352	9 / 351	0	O0/U0
	10 / 350	11 / 349	12 / 348	13 / 347	14 / 346	15 / 345	16 / 344	17 / 343	18 / 342	19 / 341	1	O1/U1
	20 / 340	21 / 339	22 / 338	23 / 337	24 / 336	25 / 335	26 / 334	27 / 333	28 / 332	29 / 331	2	O2/U2
	30 / 330	31 / 329	32 / 328	33 / 327	34 / 326	35 / 325	36 / 324	37 / 323	38 / 322	39 / 321	3	O3/U3
	40 / 320	41 / 319	42 / 318	43 / 317	44 / 316	45 / 315	46 / 314	47 / 313	48 / 312	49 / 311	4	O4/U4
	50 / 310	51 / 309	52 / 308	53 / 307	54 / 306	55 / 305	56 / 304	57 / 303	58 / 302	59 / 301	5	O5/U5
	60 / 300	61 / 299	62 / 298	63 / 297	64 / 296	65 / 295	66 / 294	67 / 293	68 / 292	69 / 291	6	O6/U6
	70 / 290	71 / 289	72 / 288	73 / 287	74 / 286	75 / 285	76 / 284	77 / 283	78 / 282	79 / 281	7	O7/U7
	80 / 280	81 / 279	82 / 278	83 / 277	84 / 276	85 / 275	86 / 274	87 / 273	88 / 272	89 / 271	8	O8/U8
	90 / 270	91 / 269	92 / 268	93 / 267	94 / 266	95 / 265	96 / 264	97 / 263	98 / 262	99 / 261	9	O9/U9
	100 / 260	101 / 259	102 / 258	103 / 257	104 / 256	105 / 255	106 / 254	107 / 253	108 / 252	109 / 251	10	O10/U10
	110 / 250	111 / 249	112 / 248	113 / 247	114 / 246	115 / 245	116 / 244	117 / 243	118 / 242	119 / 241	11	O11/U11
	120 / 240	121 / 239	122 / 238	123 / 237	124 / 236	125 / 235	126 / 234	127 / 233	128 / 232	129 / 231	12	O12/U12
	130 / 230	131 / 229	132 / 228	133 / 227	134 / 226	135 / 225	136 / 224	137 / 223	138 / 222	139 / 221	13	O13/U13
	140 / 220	141 / 219	142 / 218	143 / 217	144 / 216	145 / 215	146 / 214	147 / 213	148 / 212	149 / 211	14	O14/U14
	150 / 210	151 / 209	152 / 208	153 / 207	154 / 206	155 / 205	156 / 204	157 / 203	158 / 202	159 / 201	15	O15/U15
	160 / 200	161 / 199	162 / 198	163 / 197	164 / 196	165 / 195	166 / 194	167 / 193	168 / 192	169 / 191	16	O16/U16
	170 / 190	171 / 189	172 / 188	173 / 187	174 / 186	175 / 185	176 / 184	177 / 183	178 / 182	179 / 181	17	O17/U17
1	0 / 360	1 / 359	2 / 358	3 / 357	4 / 356	5 / 355	6 / 354	7 / 353	8 / 352	9 / 351	18	O0/U0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
31	90 / 270	91 / 269	92 / 268	93 / 267	94 / 266	95 / 265	96 / 264	97 / 263	98/ 262	99 / 261	567	O9/U9
	100 / 260	101 / 259	102 / 258	103 / 257	104 / 256	105 / 255	106 / 254	107 / 253	108 / 252	109 / 251	568	O10/U10
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
90	160 / 200	161 / 199	162 / 198	163 / 197	164 / 196	165 / 195	166 / 194	167 / 193	168 / 192	169 / 191	1636	O16/U16
	170 / 190	171 / 189	172 / 188	173 / 187	174 / 186	175 / 185	176 / 184	177 / 183	178 / 182	179 / 181	1637	O17/U17

From Table 5.6, we observe that $M = 10$ is chosen. This decision comes from the fact that the BRAM structure gets an alignment that it is easy to read data from. All the logic for indexing the BRAM structure must be designed, and by choosing $M = 10$, this task will be relatively easy with the trade-off of reduced parallelism in the search core. There is no benefit in having $M = 11$ parallel `VECMUL`s if we cannot read 11 steering vector each clock cycle. From Table 5.6, we can also observe that for each θ, the steering vector for $\varphi = 180°$ is not available. This is another trade-off with choosing a well-aligned BRAM structure. This implies that if the actual AoA has $\varphi = 180°$, we will, at best, achieve a minimum error of $1°$, hence not achieve the desired maxim error. One last remark is that $\varphi = 0°$ and $\varphi = 360°$ actually is the same angle. $\varphi = 360°$ is one full rotation on the unit circle. It is well known that

$$\begin{aligned}\cos(0°) = \cos(360°) = 1\\ \sin(0°) = \sin(360°) = 0,\end{aligned}$$

which again yield

$$\begin{aligned}\omega(\theta, 0°) &= \sin(\theta)\cos(0°) = \sin(\theta)\\ \psi(\theta, 0°) &= \sin(\theta)\sin(0°) = 0.\end{aligned}$$

The resulting steering vector will therefore be

$$\tilde{\mathbf{a}}(\theta, \varphi = 0°) = \begin{bmatrix} \cos\left(\frac{3\pi d}{\lambda}\sin(\theta)\right) \\ \cos\left(\frac{3\pi d}{\lambda}\sin(\theta)\right) \\ 0 \\ 0 \\ \cos\left(\frac{\pi d}{\lambda}\sin(\theta)\right) \\ 0 \\ \sin\left(\frac{\pi d}{\lambda}\sin(\theta)\right) \\ 0 \\ \sin\left(\frac{3\pi d}{\lambda}\sin(\theta)\right) \\ \sin\left(\frac{3\pi d}{\lambda}\sin(\theta)\right) \\ 0 \\ 0 \end{bmatrix}.$$

This means that it is unnecessary to evaluate both $\varphi = 0°$ and $\varphi = 360°$. However, it also indicates that the modified values from the function in Listing 5.2, are always 0, which means that using the same approach for these angles as for all others will therefore not yield any error. The selected approach is further explained in Section 5.3.6.

From Table 5.6, we observe that each index in each BG holds the values that can be used for two steering vectors. They both share the same value for θ, which can be found in the corresponding leftmost column. The two values for φ can be located in the same row as the index and in the same column as the desired BG. The values for φ are presented as φ_1/φ_2 where $\varphi_1 < 180°$ and $\varphi_2 > 180°$. The abbreviations O0/U0, O1/U1, ..., O17/U17 in Table 5.6 are later used for describing which angle combinations we are feeding into the search core. "O"/"U" is used for azimuth over- and under 180°, respectively. The following number, which is a number between 0 and 17, is used for describing which row for the current elevation angle to read from. As an example, "O14" is used for describing the azimuth angles

$$\overline{\varphi} = [220°, 219°, 218°, 217°, 216°, 215°, 214°, 213°, 212°, 211°].$$

From Figure 5.18, we observe that each of the available BGs can be accessed individually, allowing for only a subset of the available BGs to output data, and that individual addresses for each BG can be accessed. The use case for this will be presented in Table 5.3.6.

One drawback of the above presented method for obtaining the steering vector values is that we are not able to obtain the steering vector values for the different BLE channels. In Section 2.2, the 40 BLE channels are presented, describing how each channel has its own frequency, resulting in different wavelengths for every channel. To minimize deviation from the stored wavelength to the wavelength of the received CTE, the wavelength for the middle channel, channel 17, is used for generating the steering vector values. Due to the non-sequential order of the channels, presented in Figure 2.4, channel 17 is the middle channel with center frequency $f_{ch17} = 2.44$ GHz. A discussion of the effects of wavelength deviation in the steering vectors to the received signal is presented in Section 7.5.

The data for the BRAM are generated using Python, and are stored in a `.data`-file. The Python script creates the scaled steering vector values with a scaling factor of 10^5 and assigns an ID consisting of 8 bits to all the values. The first 4 bits indicate the BRAM ID within the BG, while the last four bits indicate the BG ID within the BRAM structure. A specific VHDL function is then written for reading the data, and generates the BRAM structure as indicated by the `.data`-file generated. All of these files can be found in git repository[2].

5.3.6 Spectral Peak Search Core

The SPS core is the top level module for this design, combining the previously described units in Section 5.3.3, Section 5.3.4 and Section 5.3.5 in order to perform the search presented in Section 5.2. This section presents the implementation of two versions for the SPS core, and they are later used to observe if it is possible to achieve the same level of accuracy by dividing the search into multiple searches compared to only one search, as presented in Section 5.2. The first version is implemented to perform only one search, and is referred to as the one-search SPS core. The second version is implemented to perform two searches, and it is referred to as the two-search SPS core. The reason for not implementing more than two searches will be discussed in the below sections. The following subsections are used to describe the architecture and the behavior of the two implemented versions.

One-Search SPS Core

The one-search SPS core performs only one search. To achieve the desired maximum theoretical error of 0.5°, the step sizes must be $\Delta\theta_1 = \Delta\varphi_1 = 1°$. With the BRAM structure presented in Section 5.3.5, these are also the lowest possible step sizes. The architecture for the one-search SPS core is presented in Figure 5.19.

[2] `https://github.com/EmLogic-Students/HDL/tree/main/sources/SPS/scripts/BRAM`

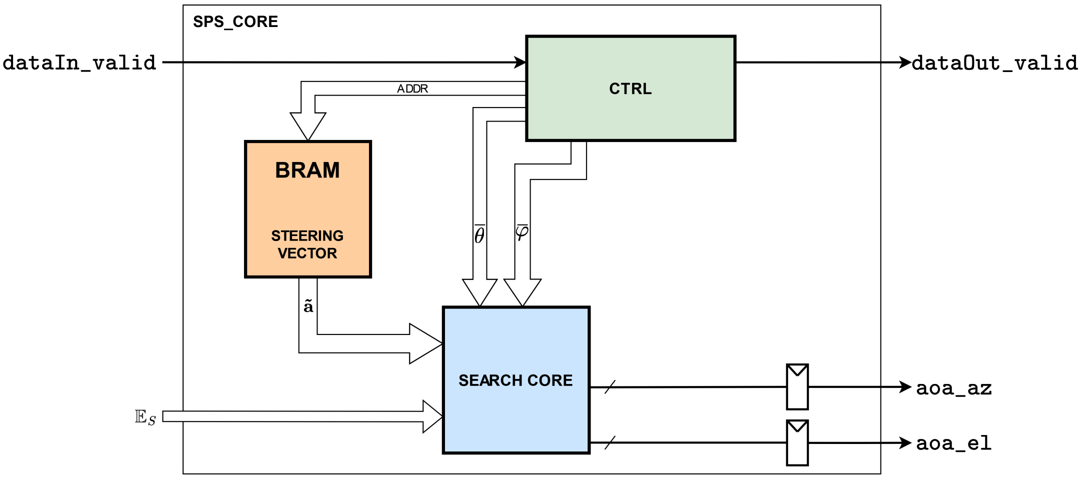


Figure 5.19: Architecture for the one-search SPS core.

The architecture is based on reading the values for $\tilde{\mathbf{a}}(\theta, \varphi)$ from the BRAM structure, and to feed them into the search core together with the correct angles. As presented in Section 5.3.2, the angle combinations for a given steering vector follow the computed value throughout the circuit to keep track of them through the pipelined architecture. This also makes it possible to feed $\tilde{\mathbf{a}}(\theta, \varphi)$ in any order, as long as the correct angles are also fed into the search core. This allows us use the special structure of the BRAM to reduce the switching activity.

In Listing 5.1, the search is performed by iterating through all values of φ incrementally before θ is incremented, and the same loop is performed again. This could be achieved for the one-search SPS core. However, by observing the BRAM structure in Table 5.6, we know that it is possible to retrieve the steering vector for two different angle combinations per index. To follow the standard iterative loop presented in Listing 5.1, the index could be incremented every clock cycle until we are at row “U17” for any given θ, before decrementing the index until “O0” is fed to the search core. This approach includes switching of addresses every clock cycle, resulting in most of the BRAM data switching too. Another alternative is to only increment the BRAM index every other clock cycle. By doing so, the first clock cycle can be used for feeding the raw data from BRAM into the search core together with the correct angle combinations ($\varphi < 180°$). For the second clock cycle, we use the function in Listing 5.2 to edit the BRAM data and feeding the results into the search core with the correct angle combinations ($\varphi > 180°$).

A typical timing diagram for the one-search SPS core is presented in Figure 5.20.

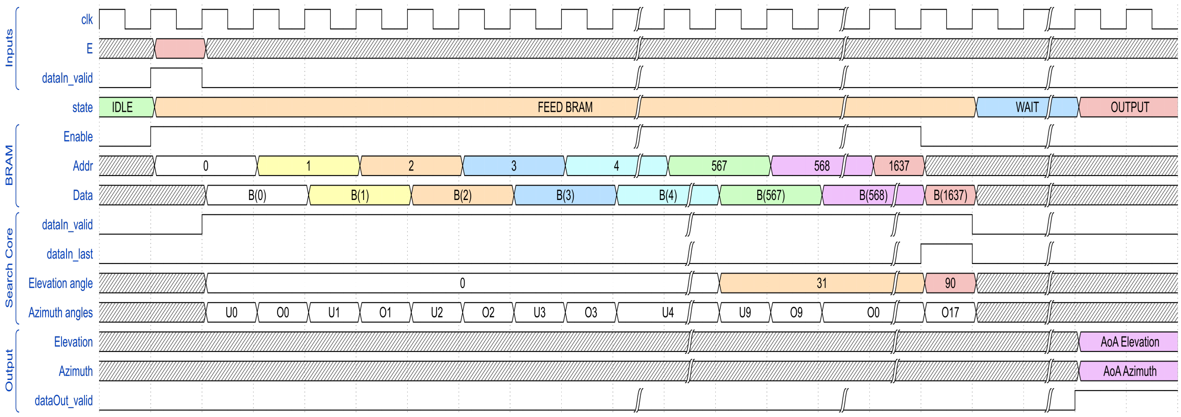


Figure 5.20: Timing diagram of the one-search SPS Core.

From Figure 5.20, it can be observed how the BRAM address is equal for all the BGs, and that it is incremented every other clock cycle. It can also be observed that the azimuth angles are updated every clock cycle. By doing so, the one-search core needs 36 clock cycles per value of θ, resulting in $91 \cdot 36 = 3276$ clock cyles to feed all BRAM data. The one-search SPS Core can be described using the state diagram in Figure 5.21. When `dataIn_valid` is set high, the `FEED BRAM DATA` state is entered, enabling the BRAM and the search core. In this state, the goal is to increment the BRAM address every other clock cycle as described above, while feeding the correct combinations of angles and BRAM data to the search core.

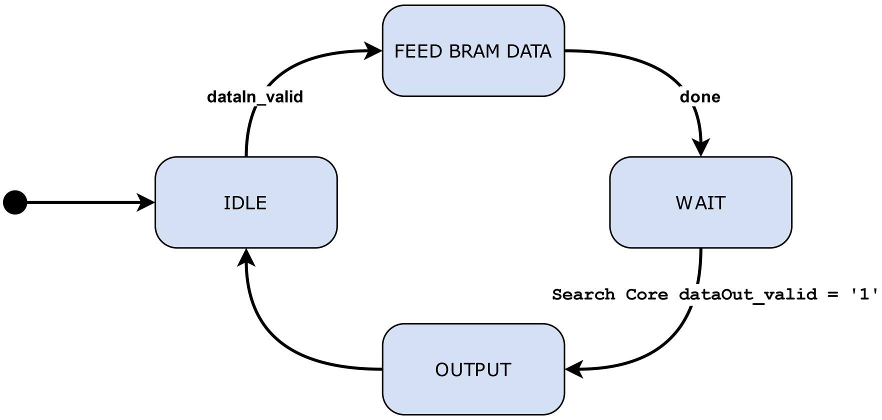


Figure 5.21: State machine of the one-search SPS Core.

While the one-search SPS core is in the `FEED BRAM DATA` state, an internal counter is enabled for keeping track of the angles to output. This counter is used for incrementing and decrementing the angles correctly. When all data are read from the BRAM structure, we need to wait for the final sets of data to be computed and evaluated inside the search core before outputting the search result. The `WAIT`-state is therefore needed, and it is entered when all data are fed to the search core. The number of clock cycles needed for allowing the search to complete after the last data is fed into search core is 14. The transition from `WAIT` to `OUTPUT` is triggered by `dataOut_valid`='1' from the search core. In the `OUTPUT` state, `dataOut_valid` in Figure 5.20 is set high, allowing the data to successfully be communicated outside of the SPS core.

An estimation of the number of clock cycles that is needed to perform a search using the one-search SPS core is presented in Table 5.7.

Table 5.7: Number of clock cycles per state for the one-search SPS core.

State	Clock cycles
`FEED BRAM`	$91 \cdot 360/10 = 3276$
`WAIT`	14
`OUTPUT`	0
Total	3290

Two-Search Core

The two-search SPS Core is implemented as an attempt to significantly reduce the number of steering vector values from the BRAM structure and computations that are needed, without reducing the precision of the search. The results in Table 5.1 indicate that performing two searches significantly reduces the number of iterations for the search without reducing the precision. Even though more search-iterations could be implemented, two search iterations are chosen as this approach most likely will perform well enough in terms of speed, and it should be able to maintain a high accuracy. The two-search SPS core is therefore designed to perform two searches with step sizes: $\boldsymbol{\Delta\theta} = \{\Delta\theta_1 = 4°, \Delta\theta_2 = 1°\}$ and $\boldsymbol{\Delta\varphi} = \{\Delta\varphi_1 = 4°, \Delta\varphi_2 = 1°\}$. The reason for choosing $\Delta\theta_1 = \Delta\varphi_1 = 4°$ is due to the layout of the BRAM structure. As presented above, M is set to be 10, and in order to maximize the throughput, we want to feed 10 new steering vectors and corresponding angles every clock cycle. This can be a bit challenging, as certain step sizes would require us to read multiple steering vector values from the same BG. As an example, if $\Delta\varphi = 10°$, one would always be required to read all steering vector values from the same BG. Using $\Delta\varphi = 10°$ would therefore make it impossible to read out the data efficiently.

Table 5.8 indicates how the BRAM structure effectively can be accessed for obtaining $M = 10$ steering vectors each clock cycle. The presented table indicates that we will need to change the BRAM addresses nine times per elevation angle. Table 5.9 is used for describing the meaning of the blue shaded cells.

Table 5.8: BRAM structure indexing for search with step sizes $\Delta\theta = \Delta\varphi = 4°$.

Common Elevation	BG 1 Azimuth	BG 2 Azimuth	BG 3 Azimuth	BG 4 Azimuth	BG 5 Azimuth	BG 6 Azimuth	BG 7 Azimuth	BG 8 Azimuth	BG 9 Azimuth	BG 10 Azimuth	Common BRAM index
2	0 / 360	1 / 359	2 / 358	3 / 357	4 / 356	5 / 355	6 / 354	7 / 353	8 / 352	9 / 351	0
	10 / 350	11 / 349	12 / 348	13 / 347	14 / 346	15 / 345	16 / 344	17 / 343	18 / 342	19 / 341	1
	20 / 340	21 / 339	22 / 338	23 / 337	24 / 336	25 / 335	26 / 334	27 / 333	28 / 332	29 / 331	2
	30 / 330	31 / 329	32 / 328	33 / 327	34 / 326	35 / 325	36 / 324	37 / 323	38 / 322	39 / 321	3
	40 / 320	41 / 319	42 / 318	43 / 317	44 / 316	45 / 315	46 / 314	47 / 313	48 / 312	49 / 311	4
	50 / 310	51 / 309	52 / 308	53 / 307	54 / 306	55 / 305	56 / 304	57 / 303	58 / 302	59 / 301	5
	60 / 300	61 / 299	62 / 298	63 / 297	64 / 296	65 / 295	66 / 294	67 / 293	68 / 292	69 / 291	6
	70 / 290	71 / 289	72 / 288	73 / 287	74 / 286	75 / 285	76 / 284	77 / 283	78 / 282	79 / 281	7
	80 / 280	81 / 279	82 / 278	83 / 277	84 / 276	85 / 275	86 / 274	87 / 273	88 / 272	89 / 271	8
	90 / 270	91 / 269	92 / 268	93 / 267	94 / 266	95 / 265	96 / 264	97 / 263	98 / 262	99 / 261	9
	100 / 260	101 / 259	102 / 258	103 / 257	104 / 256	105 / 255	106 / 254	107 / 253	108 / 252	109 / 251	10
	110 / 250	111 / 249	112 / 248	113 / 247	114 / 246	115 / 245	116 / 244	117 / 243	118 / 242	119 / 241	11
	120 / 240	121 / 239	122 / 238	123 / 237	124 / 236	125 / 235	126 / 234	127 / 233	128 / 232	129 / 231	12
	130 / 230	131 / 229	132 / 228	133 / 227	134 / 226	135 / 225	136 / 224	137 / 223	138 / 222	139 / 221	13
	140 / 220	141 / 219	142 / 218	143 / 217	144 / 216	145 / 215	146 / 214	147 / 213	148 / 212	149 / 211	14
	150 / 210	151 / 209	152 / 208	153 / 207	154 / 206	155 / 205	156 / 204	157 / 203	158 / 202	159 / 201	15
	160 / 200	161 / 199	162 / 198	163 / 197	164 / 196	165 / 195	166 / 194	167 / 193	168 / 192	169 / 191	16
	170 / 190	171 / 189	172 / 188	173 / 187	174 / 186	175 / 185	176 / 184	177 / 183	178 / 182	179 / 181	17

Table 5.9: BRAM structure indexing colors per elevation angle.

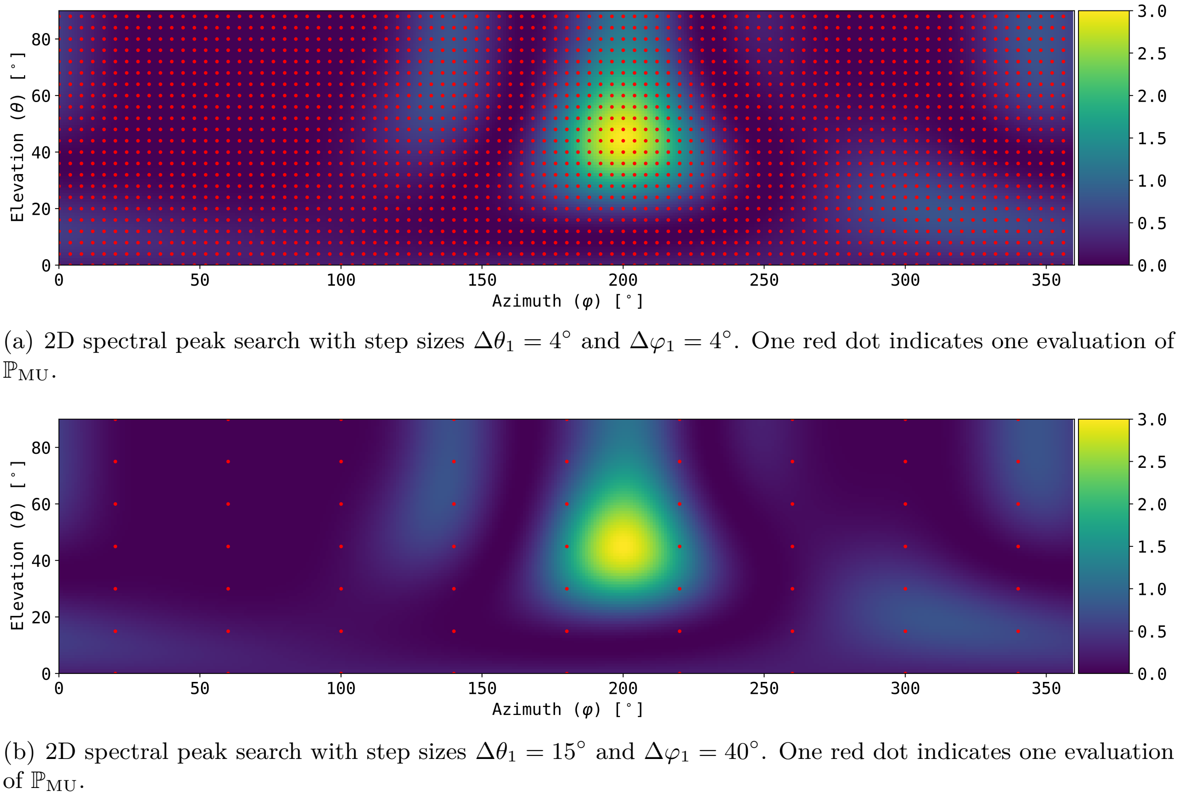
Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9

We observe that by using $\Delta\theta_1 = \Delta\varphi_1 = 4°$, the indexing is non-overlapping. Also, it can be observed that only 5 BGs are used for this search. In order to retrieve $M = 10$ values from the five highlighted BGs, the symmetric property for the steering vector, presented in Section 5.3.5, is once again used. The raw BRAM data values, with the correct $\varphi < 180°$, is passed into the search core with the positions 1 to 5, while positions 6 to 10 receive modified steering vector values, using the function presented in Listing 5.2 with $\varphi > 180°$. From Table 5.8, it can be observed that the value of θ will be the same for the data being fed to the search core per clock cycle. In order for the BRAM structure to support this type of indexing, each BG is assigned its own address input, as presented in Figure 5.18, such that the indexing in Table 5.8 can be achieved. As an example, the first BRAM data fed into the search core is retrieved from BG 1, BG 3, BG 5, BG 7, and BG 9, and the steering vectors are

$$
\begin{aligned}
\bar{\tilde{\mathbf{a}}} :& [\tilde{\mathbf{a}}(2°, 0°), \tilde{\mathbf{a}}(2°, 12°), \tilde{\mathbf{a}}(2°, 4°), \tilde{\mathbf{a}}(2°, 16°), \tilde{\mathbf{a}}(2°, 8°), \\
& \tilde{\mathbf{a}}(2°, 360°), \tilde{\mathbf{a}}(2°, 348°), \tilde{\mathbf{a}}(2°, 356°), \tilde{\mathbf{a}}(2°, 344°), \tilde{\mathbf{a}}(2°, 352°)] \\
\overline{\varphi} :& [0°, 12°, 4°, 16°, 8°, 360°, 348°, 356°, 344°, 352°] \\
\text{BRAM ADDR} :& [36, 0, 37, 0, 36, 0, 37, 0, 36, 0].
\end{aligned}
\tag{5.8}
$$

The reason why the first value for θ is set to 2° was observed when testing the high-level model. When the expected elevation angle, θ_E, was close to 0°, significant errors were observed if the start value for θ was set to 0°. From Section 3.5, we remember that it is not possible to have $\theta = 0°$ and $\varphi \neq 0°$, and errors were observed when a multi-search approach was performed using $\theta = 0°$ as start value for elevation.

Another important reason for selecting the chosen step sizes is that we reduce the risk of missing the main peak that we are searching for in the first search iteration. From Figure 5.5, we observe that the main peak is much wider than the selected step sizes. However, if large step sizes were to be used, we observe that the main peak could be missed. To illustrate this, two 2D layouts of the spectrum plot in Figure 5.5 are presented in Figure 5.22. In Figure 5.22(a), the chosen step sizes for the first search are visualized using red dots. From this figure, we can observe that the search function is evaluated multiple times within the main peak, highlighted in yellow. Figure 5.22(b) is used for visualizing how selecting too large step sizes can result in missing the main peak.



(a) 2D spectral peak search with step sizes $\Delta\theta_1 = 4°$ and $\Delta\varphi_1 = 4°$. One red dot indicates one evaluation of $\mathbb{P}_{\mathrm{MU}}$.

(b) 2D spectral peak search with step sizes $\Delta\theta_1 = 15°$ and $\Delta\varphi_1 = 40°$. One red dot indicates one evaluation of $\mathbb{P}_{\mathrm{MU}}$.

Figure 5.22: 2D visualization of how the chosen step sizes can miss the main peak.

As the core functionality for the two-search SPS core is very similar to the one-search SPS core, the two-seach SPS core can reuse most of the architecture presented in Figure 5.19. However, the two-search SPS core requires some extra logic and functionality. One of the major differences between the two versions is that the first output of the search core must be used for calculating the region for the second search, as discussed in Section 5.2. Feedback from the search core into the CTRL block can therefore be observed in Figure 5.23, which is the architecture for the two-search SPS core.

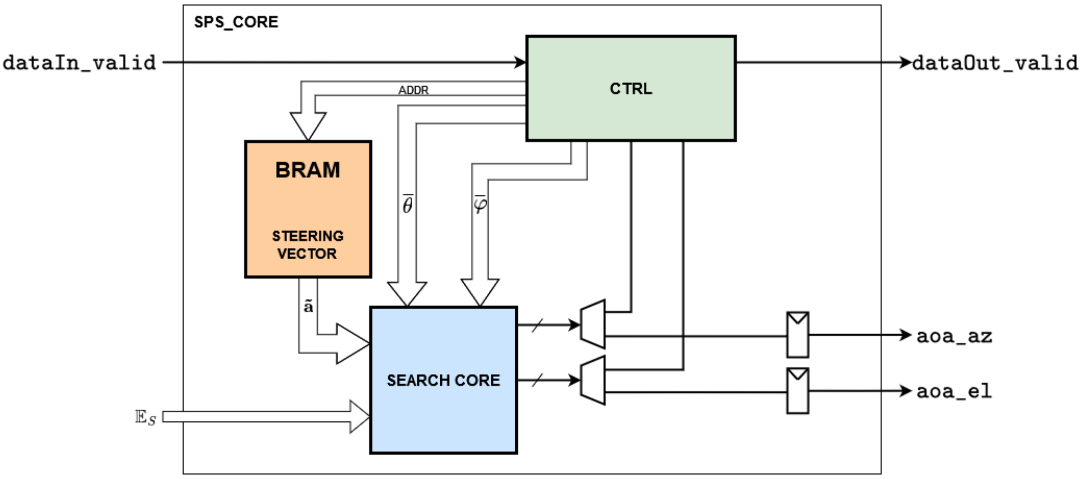


Figure 5.23: Architecture for the two-search SPS core.

This architecture is in theory very general, and could be used for performing other combinations of the search-iterations. However, most of the design within the CTRL block is highly specialized for reading the BRAM structure as presented in Table 5.8, and performing other combinations of the search-iterations would require this section of the architecture to be updated. The state machine for the two-search SPS core is more complex compared to state machine for the one-search SPS core presented in Figure 5.21. This is due to the need for processing the output from search core, and deciding where to perform the second search. The updated state diagram is presented in Figure 5.24.

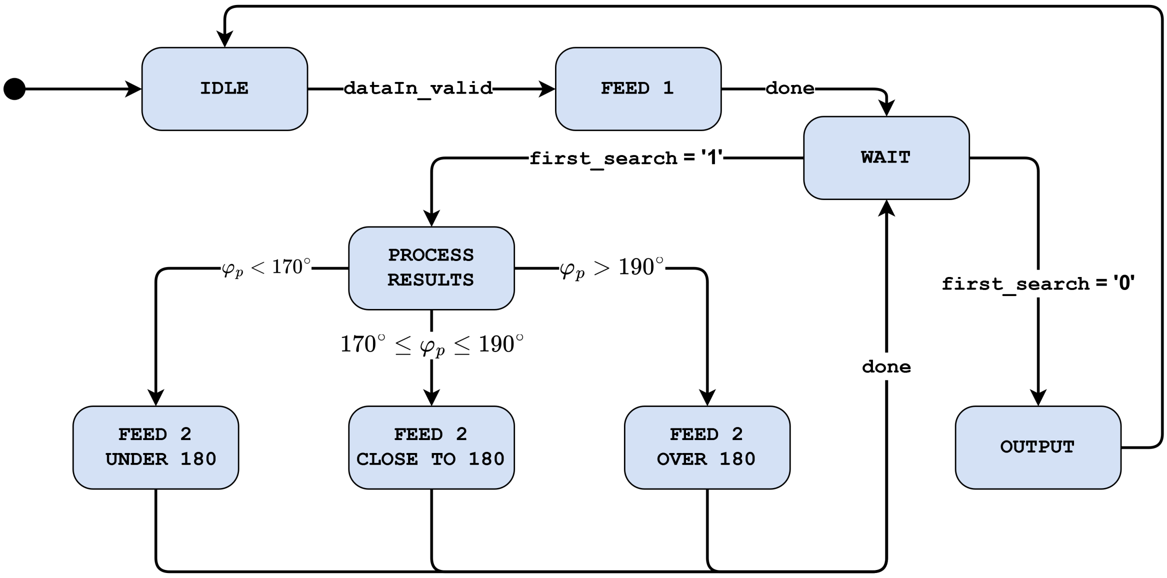


Figure 5.24: FSM for two-search SPS Core.

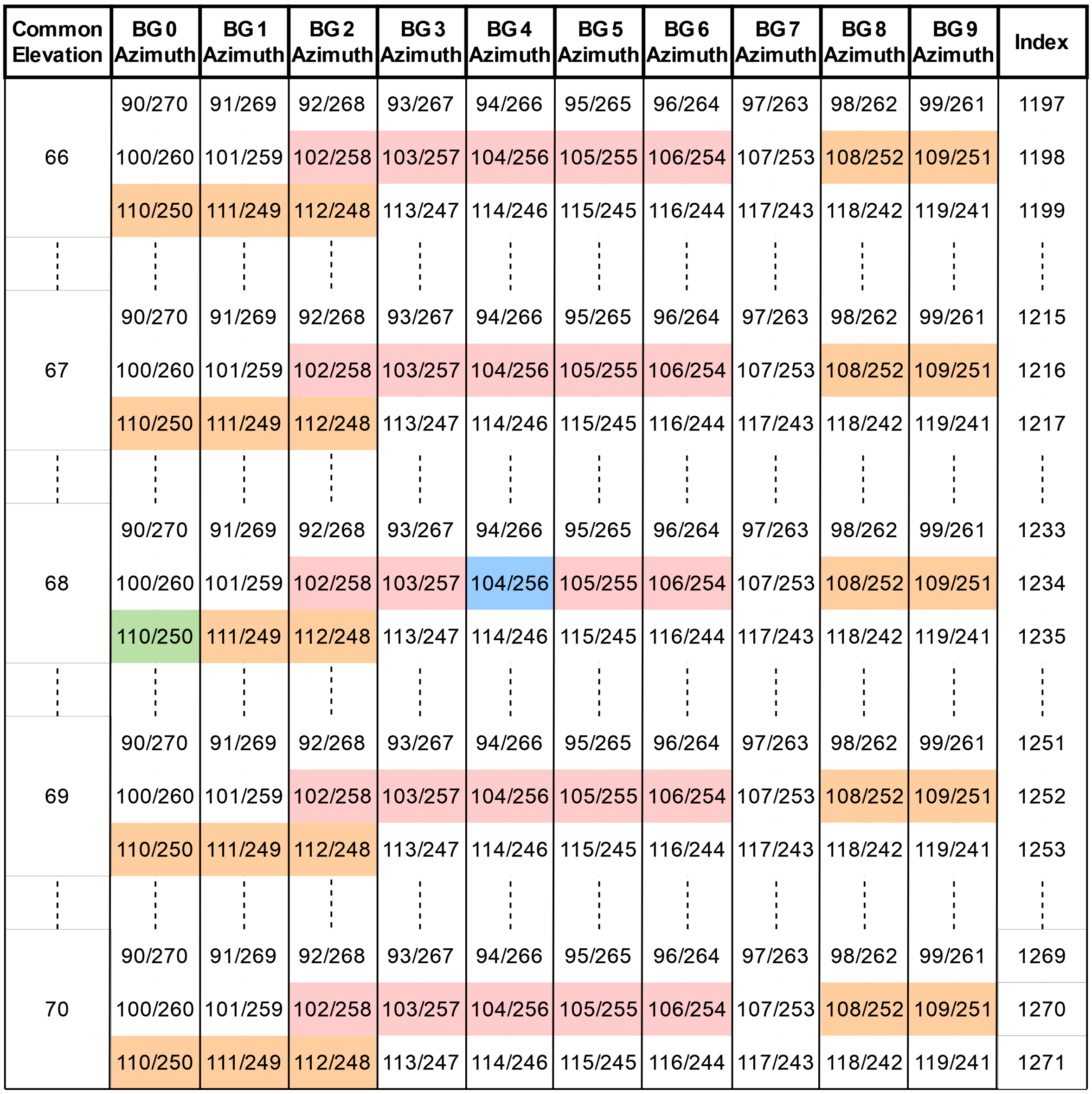
From the state diagram, we recognize some of the same behavior as described in Section 5.3.6. The search is initiated by a high value on `dataIn_valid`, changing the state from `IDLE` to `FEED 1`, which is the search with $\Delta\theta_1 = \Delta\varphi_1 = 4°$. When the desired values from the BRAM structure is read, the `WAIT` state is entered, waiting for the `dataOut_valid` output from the search core to be set high. Further transition from the `WAIT` state is conditional of the signal `first_search`, which is used to keep track of the search iteration. If `first_search` is high, which it is for the first iteration, the state transitions from `WAIT` to `PROCESS RESULT`. In this state, the output from the search core is used for obtaining the BRAM indexes to read from during the second search. How this is done is described in the following paragraphs. The signal `first_search` is set low when the transition from `WAIT` to `PROCESS RESULT` is done, such that the next time we are in the `WAIT` state, the transition will go to the `OUTPUT` state. This indicates that the search is done.

Processing the output of the search core after the first search is completed is a necessary step for obtaining the first index for the next search. From Section 5.2, we remember that the search region for the next search is centered around the peak, $(\theta_{p,1}, \varphi_{p,1})$, found in the first search. From Figure 5.24, we observe that three different transitions can be done from the `PROCESS RESULT` state: `FEED 2 UNDER 180`, `FEED 2 CLOSE TO 180`, or `FEED 2 OVER 180`. The transition is conditional of the peak azimuth, φ_p, found in the first search. The reason for the different states can be better understood by observing the BRAM structure in Table 5.6. The main reason is due to how the BRAM structure needs to be indexed for the following search.

From Section 5.2, we remember that the second search will be limited around the peak with $\pm\Delta\theta_1/2$ in the elevation region, and $\pm\Delta\varphi_1/2$ in the azimuth region. Even though this is the minimum area that is needed to search through to ensure that all possible AoAs can be found, it could also be beneficial to extend this region. Of course, searching through additional angles will increase the total search time. However, compared to the first search, the number of additional clock cycles will be very low.

Theoretically, we would only need to feed a total of $5 \cdot 5 = 25$ steering vector values into the search core for the second search. For every $\theta \in [\theta_{p,1} - 2°, \theta_{p,1} - 1°, \theta_{p,1}, \theta_{p,1} + 1°, \theta_{p,1} + 2°]$ we would need to feed steering vector values for $\varphi \in [\varphi_{p,1} - 2°, \varphi_{p,1} - 1°, \varphi_{p,1}, \varphi_{p,1} + 1°, \varphi_{p,1} + 2°]$. With $M = 10$, one would need minimum three clock cycles to feed these values $(10 + 10 + 5)$. However, with the BRAM structure presented in Section 5.3.5, this would not be feasible, as it would require us to read multiple values from the same BG simultaneously. This could be achieved by using the dual-port feature of the RAMB36E1 blocks available on the SoC as presented in Section 2.4.1. However, this is not possible as the implemented BRAM structure does not support this with the current design.

With the current layout of the BRAM structure and the search core, $M = 10$ values from the BRAM structure should be fed every clock cycle. Feeding less values will not yield faster computations. Also, when processing the output and deciding which BRAM values to read for the next search, many special cases can be encountered. Figure 5.25 is used for visualizing how the BRAM indexes for the second search could be decided for two different scenarios. In the first scenario, the peak from the first search is found for the blue cell, and the red cells must be evaluated for the second search. In the second scenario, the peak is found for the green cell, and the cells highlighted in orange must be evaluated for the second search.

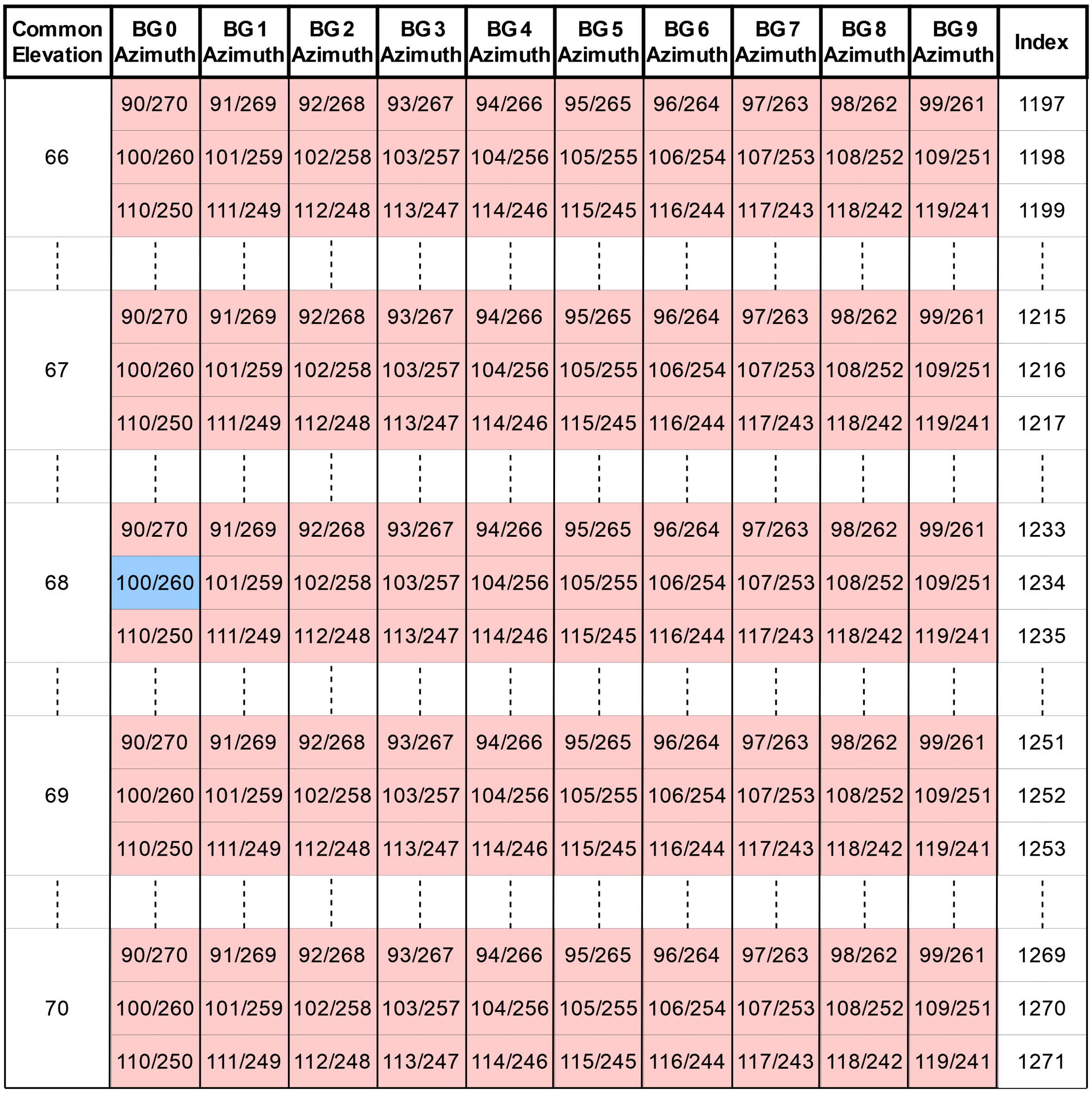


Common Elevation	BG 0 Azimuth	BG 1 Azimuth	BG 2 Azimuth	BG 3 Azimuth	BG 4 Azimuth	BG 5 Azimuth	BG 6 Azimuth	BG 7 Azimuth	BG 8 Azimuth	BG 9 Azimuth	Index
66	90/270	91/269	92/268	93/267	94/266	95/265	96/264	97/263	98/262	99/261	1197
	100/260	101/259	102/258	103/257	104/256	105/255	106/254	107/253	108/252	109/251	1198
	110/250	111/249	112/248	113/247	114/246	115/245	116/244	117/243	118/242	119/241	1199
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
67	90/270	91/269	92/268	93/267	94/266	95/265	96/264	97/263	98/262	99/261	1215
	100/260	101/259	102/258	103/257	104/256	105/255	106/254	107/253	108/252	109/251	1216
	110/250	111/249	112/248	113/247	114/246	115/245	116/244	117/243	118/242	119/241	1217
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
68	90/270	91/269	92/268	93/267	94/266	95/265	96/264	97/263	98/262	99/261	1233
	100/260	101/259	102/258	103/257	104/256	105/255	106/254	107/253	108/252	109/251	1234
	110/250	111/249	112/248	113/247	114/246	115/245	116/244	117/243	118/242	119/241	1235
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
69	90/270	91/269	92/268	93/267	94/266	95/265	96/264	97/263	98/262	99/261	1251
	100/260	101/259	102/258	103/257	104/256	105/255	106/254	107/253	108/252	109/251	1252
	110/250	111/249	112/248	113/247	114/246	115/245	116/244	117/243	118/242	119/241	1253
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
70	90/270	91/269	92/268	93/267	94/266	95/265	96/264	97/263	98/262	99/261	1269
	100/260	101/259	102/258	103/257	104/256	105/255	106/254	107/253	108/252	109/251	1270
	110/250	111/249	112/248	113/247	114/246	115/245	116/244	117/243	118/242	119/241	1271

Figure 5.25: BRAM structure indexing for the second search. The blue cell is used for indicating ($\theta_{p,1} = 68°, \varphi_{p,1} = 256°$) for one scenario, while the green cell is used for indicating ($\theta_{p,1} = 68°, \varphi_{p,1} = 110°$) for another scenario, where the indexing would be more complicated. The red/orange cells, located around the peak cells indicates which data we need to read.

The first scenario in Figure 5.25, with red and blue colors, will be straight forward to perform correctly and to feed efficiently into the search core. In this case, all the needed data for the second search are well aligned on the same index for all 5 BGs. However, in the scenario with green and orange colors, we observe that different BRAM indexes must be used, and that it will be harder to keep track of which angle combinations belongs to what data.

To avoid many edge cases and to simplify the logic for the second search, a simpler approach is taken. To reduce the number of edge cases, we want to feed whole rows from the BRAM structure into the search core. If the peak is found for the blue colored cell in Figure 5.26, feeding the whole row instead of only the red-colored cell in Figure 5.25 would not increase the time for the second search, as the search core can perform $M = 10$ parallel vector multiplications. As we do not care about which BG in the BRAM structure the peak is found from, we must feed the data from the two nearby indexes, marked with read. This approach results in needing to feed three rows per value for θ, resulting in a maximum of 15 clock cycles for the second search. This is, compared to the first search, still relatively small, making the second search efficient.



Common Elevation	BG 0 Azimuth	BG 1 Azimuth	BG 2 Azimuth	BG 3 Azimuth	BG 4 Azimuth	BG 5 Azimuth	BG 6 Azimuth	BG 7 Azimuth	BG 8 Azimuth	BG 9 Azimuth	Index
66	90/270	91/269	92/268	93/267	94/266	95/265	96/264	97/263	98/262	99/261	1197
	100/260	101/259	102/258	103/257	104/256	105/255	106/254	107/253	108/252	109/251	1198
	110/250	111/249	112/248	113/247	114/246	115/245	116/244	117/243	118/242	119/241	1199
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
67	90/270	91/269	92/268	93/267	94/266	95/265	96/264	97/263	98/262	99/261	1215
	100/260	101/259	102/258	103/257	104/256	105/255	106/254	107/253	108/252	109/251	1216
	110/250	111/249	112/248	113/247	114/246	115/245	116/244	117/243	118/242	119/241	1217
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
68	90/270	91/269	92/268	93/267	94/266	95/265	96/264	97/263	98/262	99/261	1233
	100/260	101/259	102/258	103/257	104/256	105/255	106/254	107/253	108/252	109/251	1234
	110/250	111/249	112/248	113/247	114/246	115/245	116/244	117/243	118/242	119/241	1235
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
69	90/270	91/269	92/268	93/267	94/266	95/265	96/264	97/263	98/262	99/261	1251
	100/260	101/259	102/258	103/257	104/256	105/255	106/254	107/253	108/252	109/251	1252
	110/250	111/249	112/248	113/247	114/246	115/245	116/244	117/243	118/242	119/241	1253
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
70	90/270	91/269	92/268	93/267	94/266	95/265	96/264	97/263	98/262	99/261	1269
	100/260	101/259	102/258	103/257	104/256	105/255	106/254	107/253	108/252	109/251	1270
	110/250	111/249	112/248	113/247	114/246	115/245	116/244	117/243	118/242	119/241	1271

Figure 5.26: Updated BRAM structure indexing visualization. The blue cell indicates the data for the peak found in the first search iteration, while the red cells indicate all cells read for the second search. The data for the blue cell are also included for this search.

The first step of `PROCESS RESULT` is to find the row for the $p_1 = (\theta_{p,1}, \varphi_{p,1})$ in the BRAM structure. By using one DSP slice, the *base* row index ("O0/U0") for $\theta_{p,1}$, can be found from

$$\text{Base row index} = 18 \cdot \theta_{p,1}. \tag{5.9}$$

This comes from the fact that there are 18 rows per elevation angle stored in each BG. From $\varphi_{p,1}$, we can further obtain the exact index. To find this, the offset in Equation (5.10) can be used.

$$\text{Offset index} = \begin{cases} \dfrac{\varphi_{p,1}}{10} & \text{, if } \varphi_{p,1} < 180 \\[2ex] \dfrac{360^\circ - \varphi_{p,1}}{10} & \text{, if } \varphi_{p,1} > 180 \end{cases} \tag{5.10}$$

To avoid division, which is very expensive to perform in hardware, a LUT is implemented with the same functionality as Equation (5.10). The correct index can therefore be found from the sum of Equation (5.9) and Equation (5.10). From the example in Figure 5.26, we can find the peak index for the blue cell by using Equation (5.9) and Equation (5.10) to find the base row index of $18 \cdot 68 = 1224$ and offset index of $100/10 = 10$, respectively. The correct index is therefore $1224 + 10 = 1234$. From this index, we can further adjust the index to read out the highlighted rows in Figure 5.26.

As presented, some measures are taken in order to simplify the logic for the second search. However, some special cases may still occur. These special cases have resulted in three different states for the second search, as presented in Figure 5.24. If $\varphi_{p,1} < 170^\circ$, we can safely perform the search for azimuth angles only under 180°. An example timing diagram of how the search is performed is shown in Figure 5.27.

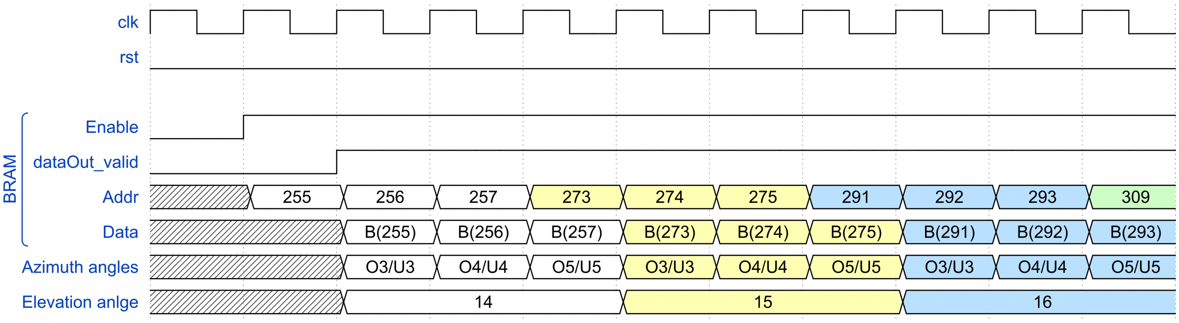


Figure 5.27: Example timing diagram when the state is `FEED 2 UNDER 180` or `FEED 2 OVER 180`.

Note that in this state, we have a corner case when $0^\circ \leq \varphi_{p,1} < 10^\circ$, as reading the data from the row before the peak index row would result in reading steering vector values for the incorrect θ- value. In this case, we only read U0 and U1 per θ, and we will only need 10 clock cycles to read all the data.

A similar search is performed if $\varphi_{p,1} > 190^\circ$. The main difference from the state where $\varphi_{p,1} < 170^\circ$ is that the BRAM data fed into the search core must be modified using the function presented in Listing 5.2. There are also some differences in how the azimuth angles are updated. However, the timing diagram is very similar for the `FEED 2 UNDER 180` state, and the example timing diagram in Figure 5.27 is used to describe the behavior of these states.

In the event of $170^\circ \leq \varphi_{p,1} \leq 190^\circ$, we would need to perform the second search including azimuth angles both over and under 180°. The technique used for reading out the BRAM data in the one-search SPS core is reused for this search. To simplify the search and avoid corner cases, all azimuth values for U16/O16 and U17/O17, and the five values of theta is fed into the search core with the corresponding data. An example timing diagram for this state is presented in Figure 5.28.

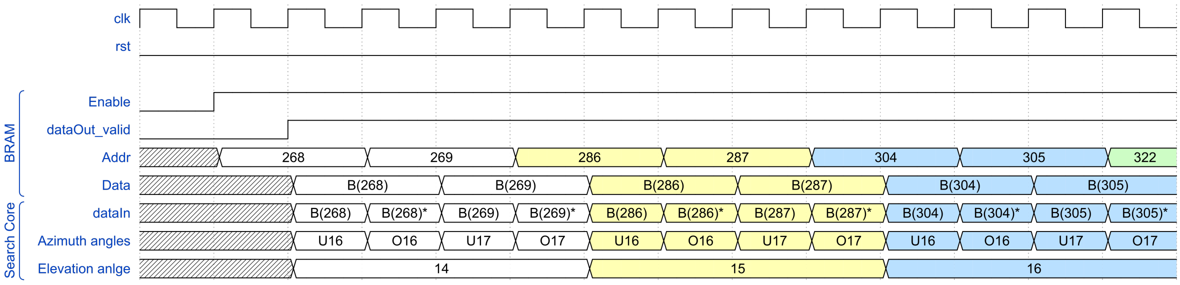


Figure 5.28: Example timing diagram when the state is `FEED CLOSE TO 180`.

The calculations performed in `PROCESS RESULT` are split over multiple clock cycles. This is done to reduce the critical path, hence increasing the maximum frequency, $f_{\max}$. When the processing of the result is complete, the two-search SPS core transitions to either of the tree available states, presented in Figure 5.24. Table 5.10 presents the estimated number of clock cycles for the available states.

Table 5.10: Number of clock cycles per state for the two-search SPS core.

State name	Number of clock cycles
`FEED 1`	$((90-2)/4 \cdot 360/4)/10 = 198$
`FEED OVER 180`	10-15
`FEED UNDER 180`	10-15
`FEED CLOSE TO 180`	20
`PROCESS RESULT`	3
`WAIT`	14

5.4 MUSIC Core

This section is used to describe the implemented design from Tommy A. Opstad and how the complete design, presented in Figure 5.1, is connected, forming the MUSIC Core. The MUSIC core is build on the design presented in this thesis and on the work presented in the thesis written by Opstad. The overall architecture for the MUSIC core is shown in Figure 5.29.

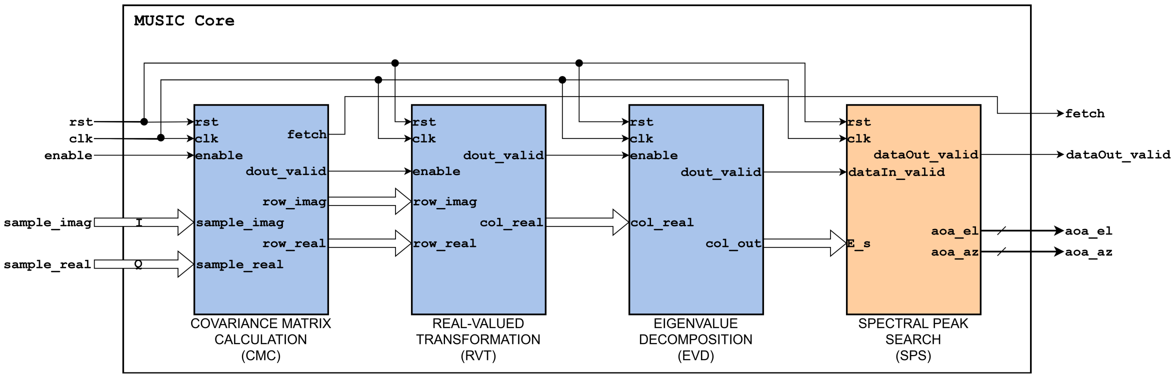


Figure 5.29: Architecture of the MUSIC core, combining the results of the work done by Tommy A. Opstad and the work presented in this thesis.

The design by Tommy A. Opstad is split into three modules. The following sections summarize his results [1].

Covariance Matrix Calculation

The CMC module is designed to perform the computations needed to obtain $\hat{\mathbf{R}}_{xx}$ from Equation (3.30). This module takes in $D = 12$ complex-valued IQ values each clock cycle, meaning that it can receive one snapshot per clock cycle. Using Gauss' method for complex multiplication, Opstad's implementation only needs three multipliers per complex multiplication, hence $3 \cdot 12 = 36$ DSP slices to perform the required covariance calculations. Using a MAC architecture, Opstad achieves the same functionality as required by Equation (3.30), and this allows us to receive N snapshots sequentially. However, as discussed in Section 4.3.3, there is no need for a scaling factor, and the division in Equation (3.30) is therefore excluded from the computations. The data-inputs of this module are also the input of the MUSIC core, and they expect signed integer values. This means that this entity, similarly as the SPS core, expects the inputs to be scaled with a number 10^n. When using a MAC architecture with signed integers, the possibility for overflow is present. To address this, Opstad added a scaling factor to the accumulated values using right shift.

Real-Valued Transformation

The RVT module combines Equation (3.34) and Equation (3.35) to transform the output of CMC, $\hat{\mathbf{R}}_{xx}$, to the real-valued covariance matrix $\mathbb{R}_{xx}$. Through a careful analysis of the transformation steps, Opstad observed one important simplification to Equation (3.34), shown in Equation (5.11).

$$\mathbf{R}_{\text{FB}} = \frac{1}{2}\left(\mathbf{R} + \mathbf{J}\mathbf{R}^{*}\mathbf{J}\right) \approx \mathbf{J}\mathbf{R}^{T}\mathbf{J} \tag{5.11}$$

This simplification yields two positive outcomes. The first one is that less resources are needed, as the matrix addition is not required. The second is that only one row needs to be received each clock cycle. The reason for feeding the rows of $\hat{\mathbf{R}}_{xx}$ is due to the transpose operator, where the rows will be converted to columns. From Equation (3.19), we remember that the rows of the leftmost matrix is multiplied with the columns of the rightmost matrix, and the functionality is therefore correct. Remembering that the matrices $\mathbf{J}$ and $\mathbf{T}$, which are needed for the transformations, only contains specific values $(0, 1, \pm j)$, Opstad achieved implementation of the transformation without using any DSP slices. This allowed increased parallelization of the SPS core.

Eigenvalue Decomposition

The final module implemented by Opstad is the EVD module, where the goal is to find the eigenvalues and eigenvectors. This is a highly complex module, and his thesis presents the supporting theory. Through his thesis, he discusses different variations of the Jacobi method, where he addresses the trade-off between resource usage and time consumption. The EVD module is implemented using the parallel Jacobi method, allowing for significantly faster computation of the eigenvectors. Through this method, the eigenvector, where the corresponding eigenvalue has the largest value, is located in the last column of the resulting matrix holding the eigenvectors. This eigenvector corresponds to $\mathbb{E}_s$, and it is further communicated with the SPS core.

Chapter 6

Test and Results

6.1 Evaluation of the High-Level Models

The high-level models used for experimenting with the algorithm during the implementation phase are also used as a reference for evaluating the performance of the hardware implementation of the SPS core. There are three main iterations of the high-level models. They are all available in the git repository that is used during this project[1]. The first iteration of the high-level model is implemented using complex-values throughout the complete algorithm. The second iteration is implemented as a real-valued, full precision algorithm, not using the scaling and limitation of precision for the steering vector, as presented in Section 5.3.5. The third and final version is the model most similar to the implemented SPS core, using a real-valued approach and reading data from precomputed memory locations. The reason for evaluating the first two versions of the high-level model is to observe if the precision is decreased for the chosen implementation, meaning that we want to observe if the precision of the search is decreased when using the real-valued approach or a quantification of the steering vector values. The last version of the high-level model is tested to observe if the hardware implementation behaves correctly. The tests are performed to evaluate three key metrics; precision, execution time and power usage. How the tests are performed, and the obtained results, are presented in the below sections.

6.1.1 Precision

For each iteration of the high-level model, two tests are performed to observe the accuracy of the high-level models. The first test is performed using only one search, while the second test uses two searches with the same step sizes as the SPS core. All tests evaluate the models for all possible angle combinations, $\theta \in [1^\circ, 90^\circ]$ and $\varphi \in [0^\circ, 359^\circ]$. The reason for excluding the tests with $\theta = 0^\circ$ is due to the fact that the signal must have $\varphi = 0^\circ$ in these cases, as presented in Section 3.5. All other values for this case are not possible. Tests with $\varphi = 360^\circ$ are also excluded as this is the same as $\varphi = 0^\circ$.

For the performed tests, two key metrics are measured. The first metric is the total number of errors. This metric is defined mathematically as

[1] `https://github.com/EmLogic-Students/High-Level-Model/tree/main/SPS`

$$\text{Number of errors} = \sum_{i=1}^{T} E_i, \tag{6.1}$$

where

$$E_i = \begin{cases} 1, & \text{if } \theta_{E,i} \neq \theta_{p,i} \text{ or } \varphi_{E,i} \neq \varphi_{p,i} \\ 0, & \text{else} \end{cases}. \tag{6.2}$$

The subscripts, (E, i) and (p, i), describe the expected output and the found peak for the i^{th} test, respectively. This metric can help us indicate if the different versions of the algorithms are likely to have many errors, or if it rarely occurs. The second metric measured is the average absolute error, finding the average absolute error using

$$\text{Average absolute error} = \frac{1}{T} \sum_{i=1}^{T} \sqrt{(\theta_{E,i} - \theta_{p,i})^2 + (\varphi_{E,i} - \varphi_{p,i})^2}, \tag{6.3}$$

where T symbolizes the total number of values to find the average of. This metric indicates the confidence level of the search. By combining the two metrics, one could also get an indication on the magnitudes of the errors, e.g., if "Number of errors" are relatively low while "Average absolute error" is relatively high, it can indicate that the few errors present has a high magnitude.

The remaining figures in this section present the plots from the tests, while Table 6.1 presents the key metrics obtained from the tests. Note that all tests are performed without additional noise added to the generated signals. A discussion on how noise influence the search is presented in Section 7.1. Testing without noise is done due to two reasons. The first being that excluding random noise from the different tests ensures us that no randomness influence the obtained results. The errors due to randomness could be minimized by performing multiple tests per simulated AoA. However, this would be very time-consuming, and it was therefore not performed for the following tests. For each of the tests presented below, $90 \cdot 360 = 32400$ simulations are performed, taking up to 24 hours per test. The second reason for testing without additional noise is to observe how well the implemented algorithm can perform under ideal conditions.

For the presented plots, the x-axis is the simulated azimuth angle, while along the y-axis, we have the simulated elevation angle. The colors indicate the absolute error, calculated by

$$\text{Absolute error} = \sqrt{|\theta_E - \theta_p|^2 + |\varphi_E - \varphi_p|^2}, \tag{6.4}$$

where (θ_E, φ_E) and (θ_p, φ_p) are the expected and found peak, respectively. For the plots with error present, the deep purple color indicates 0° absolute error. For Figure 6.1, Figure 6.2, and Figure 6.3, there are no errors present, and the color scheme is different. However, we must be aware of that only integer values for the AoA are tested, and as the minimum step size for the search is 1° for both azimuth and elevation due to limited memory, we will observe a maximum of 0.5° absolute error if the AoA contains real numbers.

Table 6.1: High-level model metrics for the performed tests. CV: complex-valued, RV: real-valued.

Test description	Number of errors	Average error [°]
CV Full precision one-search	0	0
CV Full precision two-search	0	0
RV Full precision one-search	0	0
RV Full precision two-search	16	0.0005
RV Constrained precision one-search (5 decimals)	748	0.02
RV Constrained precision two-search (4 decimals)	2609	0.09
RV Constrained precision two-search (5 decimals)	748	0.02

Complex-Valued High-Level Model

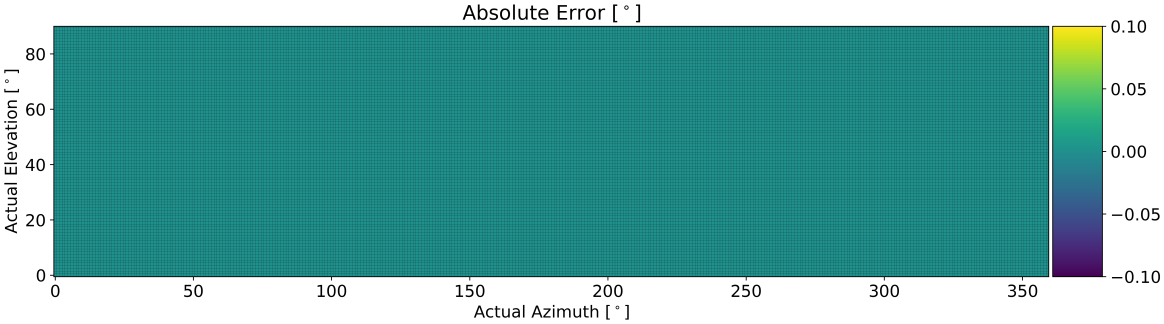


Figure 6.1: CV, full precision one-search high-level model absolute error plot.

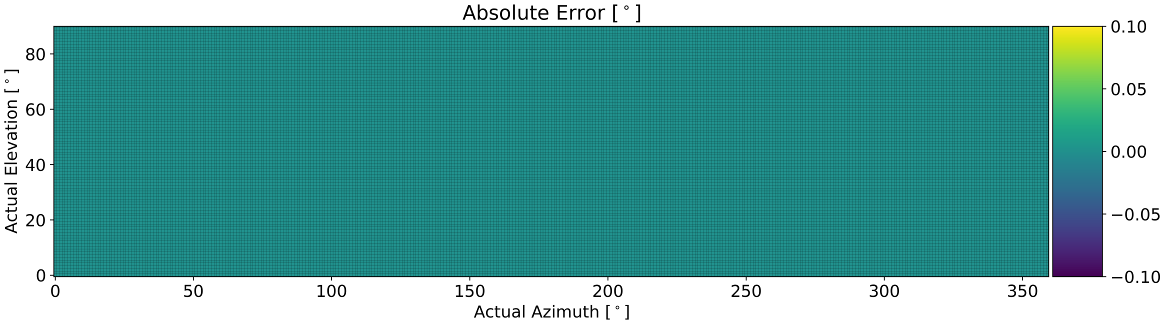


Figure 6.2: CV, full precision two-search high-level model absolute error plot. This plot indicates that there are no errors.

Full precision, Real-Valued High-Level Model

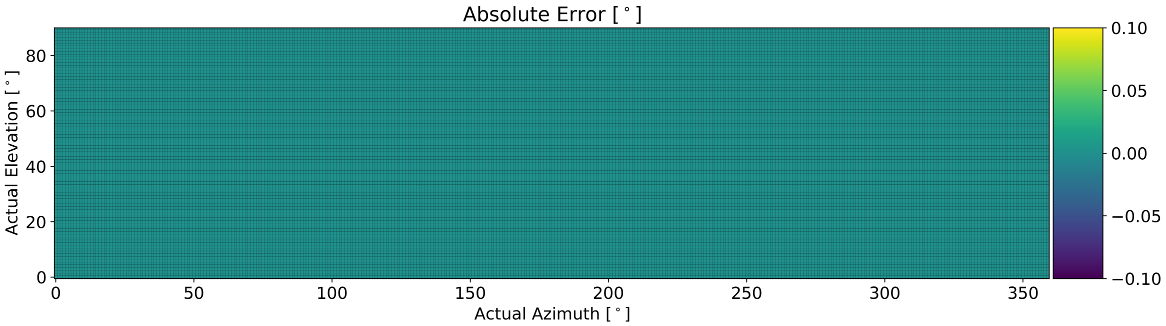


Figure 6.3: RV, full precision one-search high-level model absolute error plot.

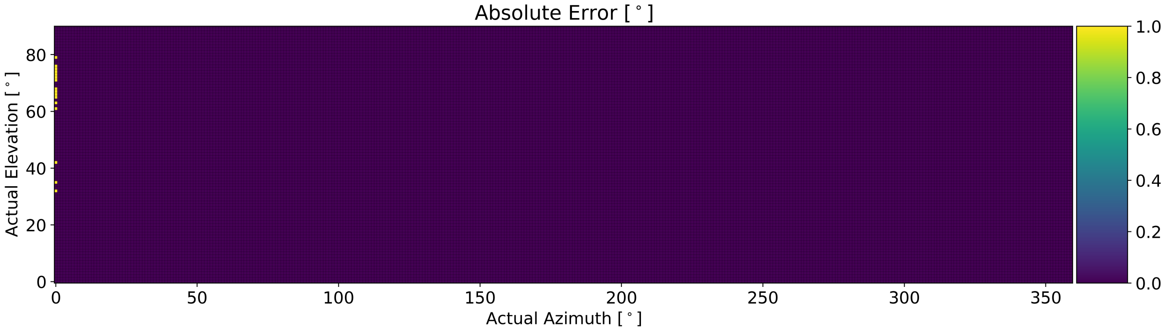


Figure 6.4: RV, full precision two-search high-level model absolute error plot.

Contained Precision, Real-Valued High-Level Model

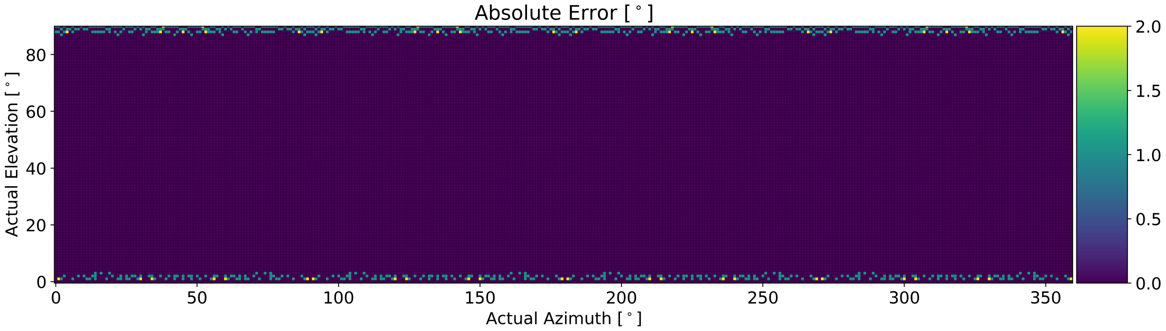


Figure 6.5: RV, constrained precision one-search high-level model absolute error plot. The precision used for this test is 5 decimal places, i.e., the same as the implemented hardware accelerator.

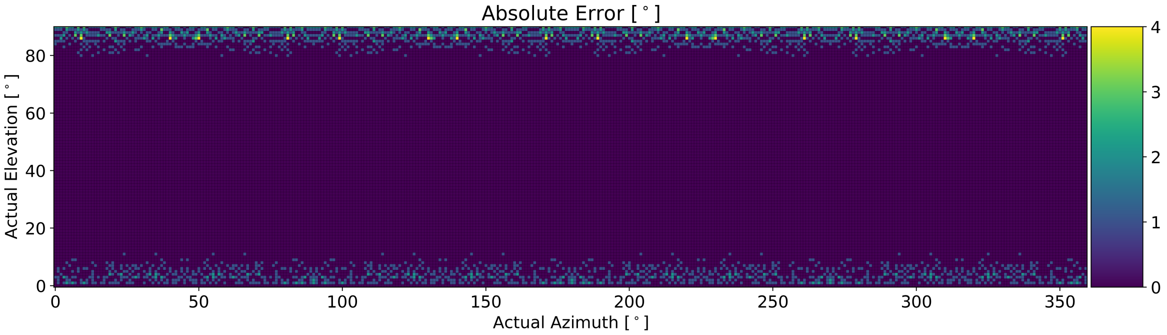


Figure 6.6: RV, constrained precision two-search high-level model absolute error plot. The precision used for this test is 4 decimal places.

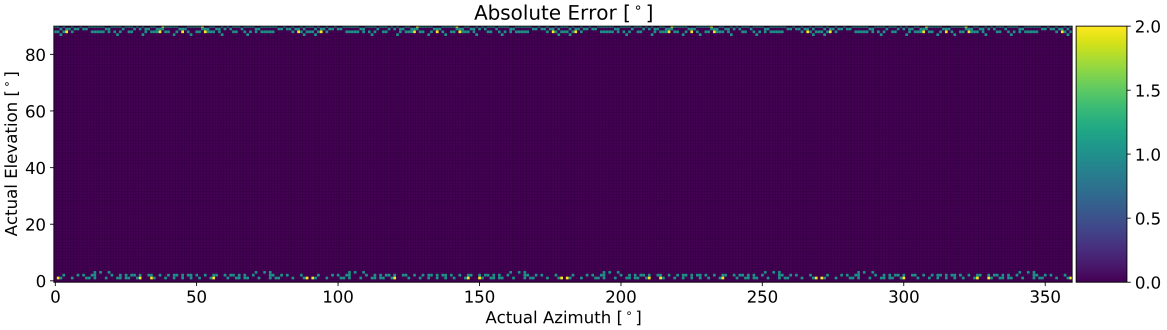


Figure 6.7: RV, constrained precision two-search high-level model absolute error plot. The precision used for this test is 5 decimal places, i.e., the same as the implemented hardware accelerator.

Time- and Power Statistics of High-Level Model

As presented in Section 1.2, one of the main objectives for this thesis is to compare a software-based implementation of the MUSIC algorithm to the hardware accelerated MUSIC algorithm in terms of time- and power consumption. To allow for this discussion, timing- and power analysis is performed on the implemented high-level models. The high-level models used for the precision plots in Section 6.1.1 are implemented in a very ineffective way, making them unusable for comparing time- and power. These models are implemented with much flexibility, adding support for testing- and experimenting with multiple searches and different step sizes. In this context, new and simplified high-level models were developed, implementing only the necessary functionality for performing the SPS. In addition to creating minimalistic Python versions of SPS, one version is also written in C. C is known for its fast execution time, and a model is implemented how fast a software-based algorithm can be executed. All of these versions are verified for a limited number of AoAs to ensure that the correct behavior is present. All the minimalistic high-level models are available in the git repository[2]. For the simplified high-level models, the SPS is performed 1000 times, and the reported timing result is found from the average of these tests. The obtained timing results are presented in and Table 6.2.

[2] `https://github.com/EmLogic-Students/High-Level-Model/tree/main/SPS`

Table 6.2: Timing statistics for the high-level models.

High-Level Model	Measured time [µs]
Python complex-valued one-search	154 272.85
Python complex-valued two-search	10 531.02
Python real-valued one-search	57 369.15
Python real-valued two-search	4 023.52
C real-valued one-search	181.4
C real-valued two-search	17.4

To measure the power for each of the performed tests, the best solution was to use Open Hardware Monitor [38], an open-source application that reports, amongst other things, the CPU power on the computer. To reduce other tasks interfering with the tests, the computer used for performing these tests was set to airplane mode, disabling as many tasks as possible. The charger of the computer was unplugged, and no external peripherals are connected. The measurements presented in Figure 6.8 is obtained by running the SPS one-search version repeatedly for some time, allowing the CPU power to stabilize. Approximate average power consumption is read from the CPU power chart in Open Hardware Monitor, and it is presented in Table 6.3. For further discussion of these results, the reference power, marked with the red line in Figure 6.8, is subtracted from the measurements of each of the performed power tests.

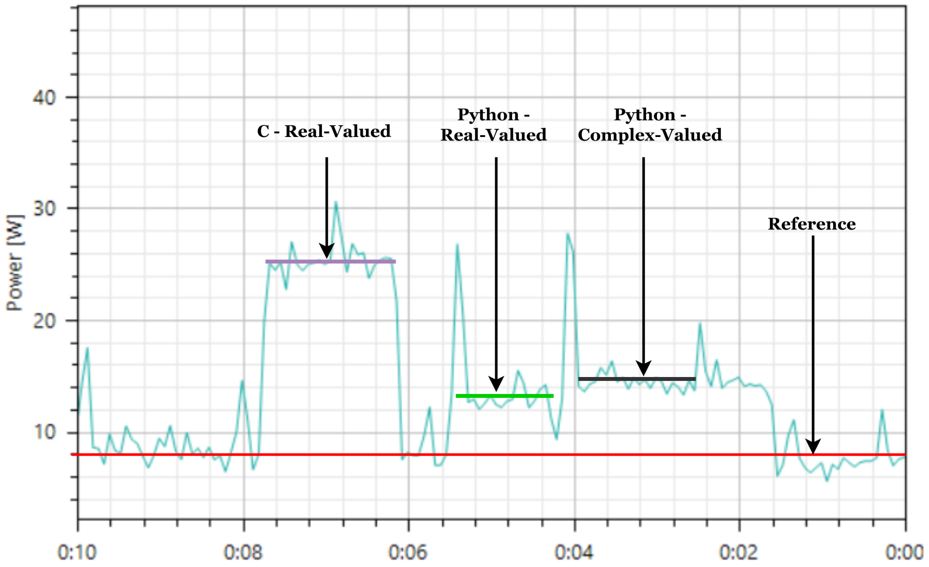


Figure 6.8: Results from observing the reported CPU power while continuously running the high-level models on a laptop. The data is obtained by using Open Hardware Monitor [38].

Table 6.3: Power usage on a computer running the implemented high-level models. All measurements are performed using Open Hardware Monitor [38].

High-Level Model	Approximate Average Power [W]
Reference	8
C Real-Valued	26
Python Real-Valued	13
Python Complex-Valued	15

6.2 Test and Verification of SPS Core

The implemented design presented in Chapter 5 is verified through multiple steps. All modules inside the SPS CORE are verified and synthesized individually before both versions of the SPS core, presented in Section 5.3.6, are tested. The following subsections present the methods used for testing the design and the obtained results. All files that are generated and written for testing and verifying the implemented design can be found in the HDL git repository[3].

Verification of Vector Multiplication Unit

The `VECMUL` unit provides a central functionality for the SPS core, and verifying correct behavior is therefore essential. This is the goal for the testbench (TB), presented in Figure 6.9.

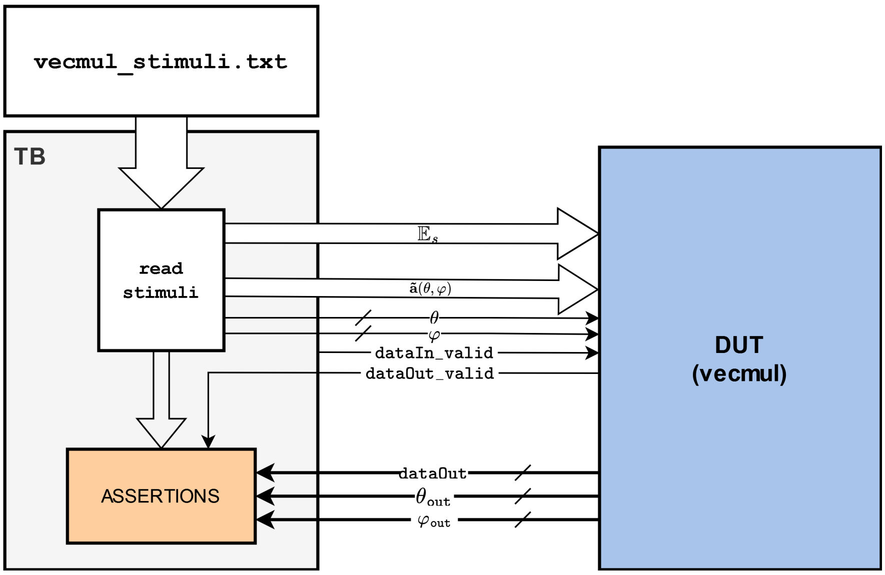


Figure 6.9: Block diagram of the TB used for verification of `VECMUL`.

The TB is based on reading a text file containing stimuli and expected results, named `vecmul_stimuli.txt`. The text file is generated using a Python script. Using `NumPy` [39], two random vectors with $D = 12$ elements are generated. Each generated value is limited within $[-2^{17}, 2^{17} - 1]$, which is the achievable range when using 18 bits per value. The two

[3] `https://github.com/EmLogic-Students/HDL`

vectors are multiplied, and the absolute value is taken of the result. This value will be used by the TB to verify that VECMUL gives the exact same output. To verify that VECMUL is able to shift the input angles throughout the pipelined architecture, as presented in Figure 5.11, a total of $91 \cdot 361 = 32\,851$ random vector multiplications are generated and stored together with the individual angle-pairs to input to the SPS core. The expected result is also stored in this file. This structure of the generated data is documented in `vecmul_stimuli_gen.py`, and can be located in the git repository.

When `dataOut_valid` from the DUT is set high, the TB checks if the `dataOut` and the outputted angles are equal to the expected outputs. The TB keeps track of the number of outputted values, which is used for reading the correct data from the stimuli file. The TB feeds one new test into the Device Under Test (DUT) each clock cycle. This is done to verify that the pipelining architecture works as intended.

When running the TB, none of the assertions are triggered for the 32 851 random tests, indicating that the module is implemented as intended.

Synthesis Reports of Vector Multiplication Unit

The VECMUL module is synthesized using Vivado. The reported maximum frequency is $f_{\text{max}} =$ 184MHz.

Table 6.4: Synthesis report in terms of resource usage for the VECMUL unit.

Resource	Number of resources used
DSPE48E1	16
LUT	97
RAMB36E1	0
Registers	246

Table 6.5: Synthesis report in terms of power usage for the VECMUL unit. All power reports are generated with the default parameters for Vivado, presented in Table A.1.

Power Type	Value [mW]
Static power	103
Dynamic power	98
Total power	201

Verification of COMP and COMP2

Comparing the outputs from the $M = 10$ VECMUL entities is the second key functionality for both versions of the SPS core. The TB made to verify both COMP and COMP2 is developed using the same approach as for VECMUL, and the block diagram for the TB is presented in Figure 6.10.

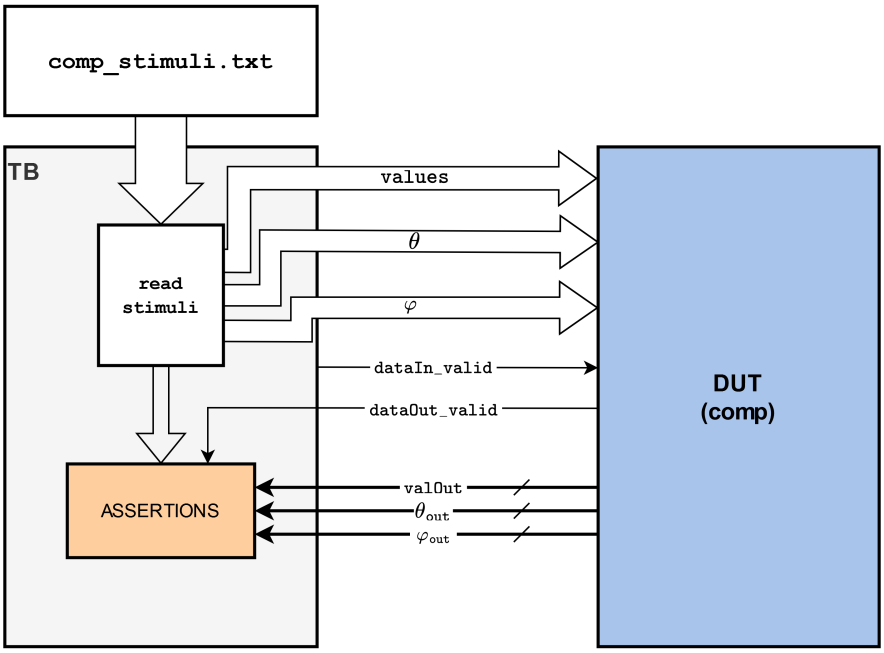


Figure 6.10: Block diagram of the TB used for verification of the `COMP` entity.

The goal for the TB is to ensure that the intended behavior is implemented. For verifying this, the strategy is to generate stimuli using the python script `comp_stimuli_gen.py` and storing the generated data in a text file named `comp_stimuli.txt`. Using `NumPy`, $M = 10$ random integer values, together with random angle values combinations (θ, φ), are generated per test. From the bit width of 18 for the steering vector values, we know that the expected bit width of the values fed to `COMP` is $2 \cdot 18 = 36$, and that they will be unsigned. The expected value is also found when generating the data, and it is stored together with the stimuli, allowing the TB to read both the stimuli and the expected result. The TB feeds new stimuli each clock cycle and asserts the output with the expected data for every clock cycle `dataOut_valid` is set high. This strategy also allows us to verify that the pipelining architecture works as intended.

The implemented TB runs 500 tests without any errors, indicating that the DUT works as intended. It is sufficient to verify the `COMP` entity, as a verified behavior of this entity implies correct behavior of the `COMP2` units inside. Table 6.6 holds the synthesis results for the `COMP` entity in terms of resources, while Table 6.7 presents the power report for the unit. It is verified that one full round of comparison and output takes five clock cycles and that the pipelining works as intended.

Synthesis Reports for Comparison Unit

The `COMP` module is synthesized using Vivado. The reported maximum frequency, $f_{\text{max}} = 276\text{MHz}$.

Table 6.6: Synthesis report in terms of resource usage for the COMP unit.

Resource	Number of resources used
DSPE48E1	0
LUT	367
RAMB36E1	0
Registers	701

Table 6.7: Synthesis report in terms of power usage for the VECMUL unit. All power reports are generated with the default parameters for Vivado, presented in Table A.1.

Power Type	Value [mW]
Static power	106
Dynamic power	102
Total power	208

Verification of Search Core

The search core is mainly build using $M = 10$ VECMUL entities and one COMP entity. With these already verified, we know that the core functionality works. However, the search core introduces some new logic, as well as a specific input combination of dataIn_valid and dataIn_last, for indicating that the search should be completed. Also, the connections of the instantiated VECMUL entities to the COMP entity must be verified. A new TB is therefore implemented, where the goal is to verify that the search core behaves as expected. The block diagram for this TB is presented in Figure 6.11.

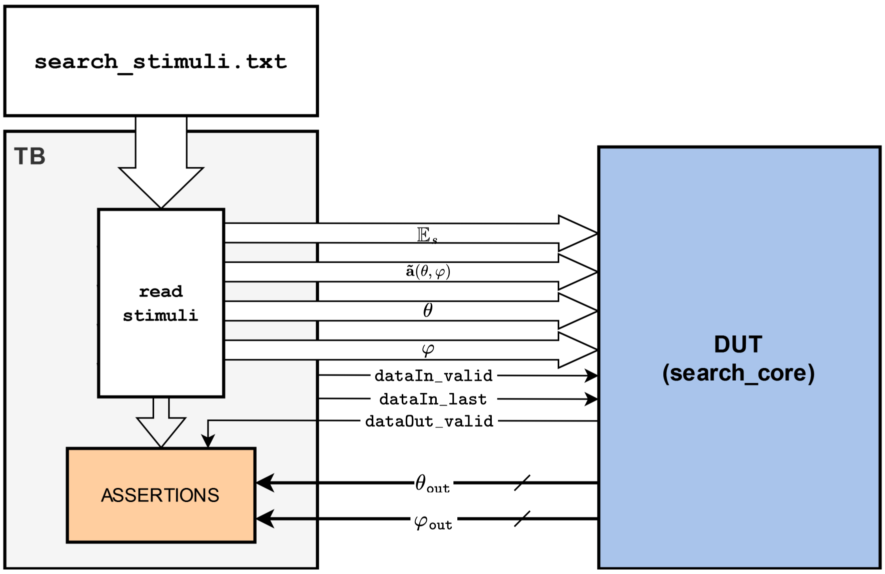


Figure 6.11: Block diagram of the TB used for verification of the search core.

Compared to the TB for VECMUL and COMP, this TB does not feed new stimuli each clock cycle. From Section 5.3.2, we remember that the search core must be able to receive a random number of inputs, and that dataIn_valid and dataIn_last are used communicating that there are no more inputs to feed, and that the peak angles can be made available when the search is complete. For each test, one random vector of integer values is generated, simulating $\mathbb{E}_s$. To simulate reading data from the BRAM, random steering vector and corresponding random angles are also generated. A Python script is written to generate this data, as well as precomputing the expected values for the search. All the random generated steering vectors are multiplied with $\mathbb{E}_s$, and the corresponding angle combinations of the highest product are saved to the stimuli file, `search_stimuli.txt`.

A total of 150 tests, all including 150 steering vectors, are fed into the DUT. This means that the TB feeds the steering vectors over 15 clock cycles per test. When running the TB, no assertions are triggered, indicating that the correct output is received, and that the DUT works as intended.

Synthesis Reports for Search Core

The search core is synthesized using Vivado. The reported maximum frequency, $f_{\text{max}} = 184\text{MHz}$.

Table 6.8: Synthesis report in terms of resource usage for the search core unit.

Resource	Number of resources used
DSPE48E1	160
LUT	1547
RAMB36E1	0
Registers	3508

Table 6.9: Synthesis report in terms of power usage for the search core. All power reports are generated with the default parameters for Vivado, presented in Table A.1.

Power Type	Value [mW]
Static power	111
Dynamic power	424
Total power	535

Test of SPS Core

The SPS core is the top level of the design presented in this thesis, combining the previously verified entities and the BRAM structure. As presented in Section 5.3.6, two versions of the SPS core are implemented, both with the same expected behavior. It is therefore sufficient with one TB for testing both of the implementations. The block diagram for the TB is presented in

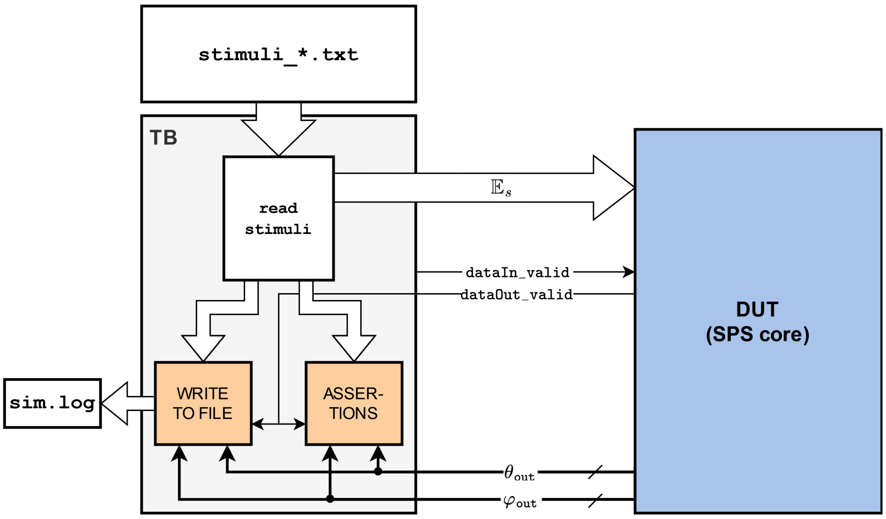


Figure 6.12: Block diagram of the TB used for both implemented versions of the SPS core.

The tests for the SPS core are generated with a similar strategy as for the other TBs. A Python script is written to generate stimuli. For each test, two random integer values, simulating θ and φ are generated. The high-level model, used for experimenting and verifying the algorithm, is firstly used for finding $\mathbb{E}_s$, using the NumPy library [39]. In theory, this should be sufficient, as we only need to feed $\mathbb{E}_s$ to the SPS core. However, to ensure that the stimuli is correctly generated, the search is also performed using the high-level model when generating stimuli, letting the developer know if there are any possible issues with the stimuli generation.

The assertions written for this TB do not require the output to be exactly equal to the expected results. From the high-level tests with constrained precision, presented in Table 6.1, we observe that we must expect some errors. Especially for $\theta > 80°$, the high-level model introduces more errors, and the assertions are adjusted to be less strict for these values.

The TB for the SPS core has three tasks. These are to apply stimuli when the SPS core is ready, assert the output with the expected values generated by the Python script, and write the results to a log-file. The latter task is added, as it makes it easy to visualize the performance in terms of precision for the SPS core.

The one-search SPS core is tested using one test type only. Stimuli are generated for $90 \cdot 360 = 32400$ tests, where each test has a unique combination of expected AoA. This is identical to the one done for the high-level model, presented in Section 6.1, and the obtained results will be used for discussing the performance in Chapter 7. The obtained results from the tests are read from the log-file, and the results are visualized in Figure 6.13.

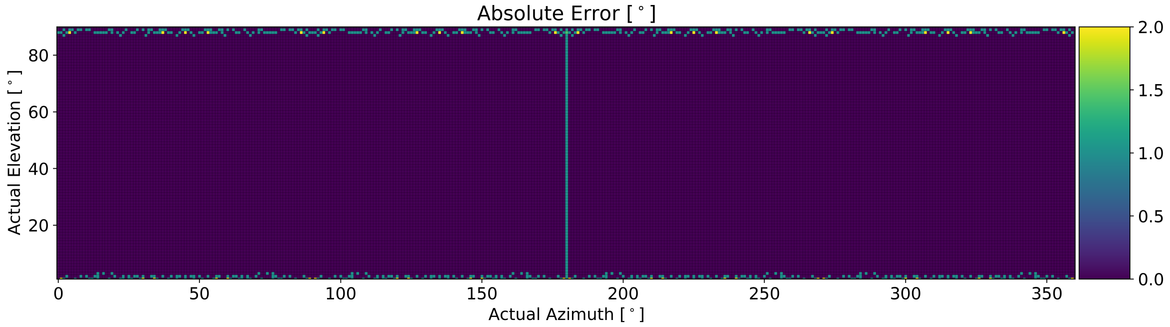


Figure 6.13: Precision plot for the one-search SPS core.

The two-search SPS core is tested in multiple steps. To ensure that the logic for the different states, `FEED 2 UNDER 180`, `FEED 2 OVER 180` and `FEED 2 CLOSE TO 180`, is correct, three stimuli sets are generated, constraining the randomness of the AoA to be within the different regions. The TB is adjusted to read one of the available stimuli files per run, and the correct behavior for all cases is observed. This made it easier to verify that the two-search SPS core works as intended, and it implies that the full tests should be able to give a correct representation of how well the algorithm can perform. The final step for testing the two-search SPS core is the same test as used on the high-level model and the one-search SPS core. The obtained results from this test are presented in Figure 6.14.

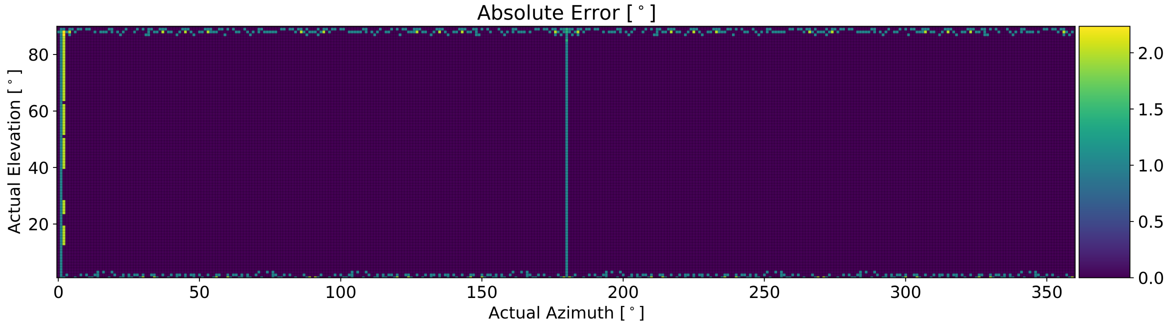


Figure 6.14: Precision plot for the two-search SPS core.

The key metrics for precision of the two implemented versions are presented in Table 6.10.

Table 6.10: Precision metrics for the RTL implementation of the SPS core.

Test description	Number of errors	Average error [°]
One-search SPS core	756	0.02
Two-search SPS core	894	0.03

The timing statistics for the two versions of the SPS core are also observed, and are presented in Table 6.11.

Table 6.11: Timing statistics for the two versions of the SPS CORE.

State	Two-Search SPS Core		One-Search SPS Core	
	# Clock cycles	Time used [µs]	# Clock cycles	Time used [µs]
Feed 1	209	1.47	3184	20.41
Wait	14	0.099	14	0.09
Process result	4	0.028	-	-
Feed 2	15-20	0.11-0.14	-	-
Total	256-261	1.8-1.83	3198	20.5

A more complete power report, summarizing the SPS core, is presented in Table 6.12.

Table 6.12: Summary of timing and power usage for each of the implemented modules. All power reports are generated with the default parameters for Vivado, presented in Table A.1.

Module	$f_{\max}$	Static power [mW]	Dynamic power [mW]
VECMUL	184.5	106	98
COMP2	N/A	105	38
COMP	276.3	106	102
Search core	184.5	111	424
One-search SPS core	156	118	482
Two-search SPS core	142	122	598

6.3 MUSIC Core

The complete MUSIC core is not fully tested. The high complexity module implemented by Opstad, EVD, does not function properly, resulting in incorrect eigenvectors. As presented in Section 3.6.3 and Section 5.3.6, the SPS core needs the signal subspace eigenvector. When incorrect eigenvectors are received, we will not be able to perform a successful search. A simple TB is created by Opstad, and can be found in the git repository. However, as we expect incorrect behavior, no thorough tests are performed. Opstad has proved that the chosen method of implementation works by using a high-level model [1], indicating that it should work. He has also successfully implemented the module for smaller array sizes. If this module is implemented correctly, the complete MUSIC core could be verified and tested with a similar approach as the above describe tests.

Implementation Reports for MUSIC Core

For the synthesis and implementation of the top level MUSIC core, Vivado is used. No errors are reported while running implementation, indicating that the design presented in this report and the design by Tommy A. Opstad are compliant with each other. Reports from the successful implementation are presented in the below tables. Table 6.13 presents the individual module device utilization and the total utilization of the MUSIC core, Table 6.14 presents the maximum frequency and timing statistics for the design, while Table 6.15 presents the power reports obtained for the design. The power reports are first generated running synthesis for each of the submodules in the MUSIC core, where all modules are running at its maximum operating frequency. The MUSIC core data represents the final power report, running at 100 MHz.

Table 6.13: MUSIC core device utilization report for ZYNQ XC7Z020. Parts of these results are retrieved from [1].

Module	Look-Up Tables	Registers	DSP Slices	BRAM Tile	F7 Mux
CMC	3 978	4761	36	0	0
RVT	2 511	4 935	0	6	360
EVD	17 190	5 181	0	0	0
One-search SPS core	3 588	5 114	160	120	0
Two-search SPS core	6 482	6 042	161	120	640
Total (Using two-search SPS core)	30 161	20 829	197	120	1000
Device utilization (Using two-search SPS core) [%]	56.7	19.6	89.6	85.7	3.8

Table 6.14: MUSIC core timing report for ZYNQ XC7Z020. Parts of these results are retrieved from [1].

Module	Max clock frequency [MHz]	Number of clock cycles	Execution time [µs]
CMC	160	48	0.3
RVT	100	168	1.68
EVD	100	153	1.53
One-search SPS	156	3198	20.5
Two-search SPS	142	261	1.83
Total (two-search/one-search)	-	630/3567	5.34/24.01

Table 6.15: MUSIC core power report for ZYNQ XC7Z020. All power reports are generated with the default parameters for Vivado, presented in Table A.1. Parts of these results are retrieved from [1].

Module	Static power [mW]	Dynamic power [mW]
CMC	110	335
RVT	106	78
EVD	119	836
One-search SPS	118	482
Two-search SPS	122	598
MUSIC core (Using two-search SPS core)	138	1319

Chapter 7

Discussion

This chapter aims to discuss the obtained results presented in Chapter 6, with respect to the objectives presented in Section 1.2. A discussion of how a potential complex-valued architecture would influence the obtained results is given. Finally, a recommendation of further improvements and future work is given.

7.1 Precision

When comparing the precision plots for the two versions of the hardware implementation for the SPS core, presented in Figure 6.13 and Figure 6.14, to the results from the high-level model in Section 6.1, we observe similarities, indicating that the hardware implementations follow the desired behavior. We firstly observe, from Table 6.1 and Table 6.10, that the one-search SPS core only introduces 8 more total errors than the corresponding high-level model. As the error for $\varphi = 180°$ is always 1°, this is somewhat surprising, as we should expect around 90 more total errors, where one additional error should be observed for each θ_E. To remove this constant error, the steering vector values could be stored in the BRAM structure. From Figure 5.15, we observe that the BRAM structure does not fill all individual BRAMs fully, leaving room to store the steering vectors for $\theta \in [0°, 90°]$ and $\varphi = 180°$. Doing so would however increase the required time for the search, as there is no easy way to add this vector into the current vectors that are being passed into the search core.

Both the high-level simulations and the implemented hardware versions indicate that the search becomes more unstable for $\theta_E \leq 4°$ and $\theta_E \geq 87°$ when using constrained precision. These errors are not present in the full precision high-level simulations, presented in Figure 6.1, Figure 6.2, Figure 6.3, and Figure 6.4. This indicates that the quantization of the steering vector- and eigenvector values yields an increase in error. The problem occurs in the given regions, and the reason can be observed from the arguments in the steering vector as θ approaches the limits. As presented in Section 5.3.5, 5 decimal precision is the maximum level of precision achievable with the decision of scaling all values with 10^5 and storing them as a signed integer. Using an FPGA with more available BRAM could therefore reduce the observer errors. If 32 bits were used for each value, we could obtain a precision of 9 decimals, that are significantly more than the implemented precision of 5 digits. However, this would require $90 \cdot 360 \cdot 12 \cdot 32$ b $\approx$ 1.5MB of memory, which would be expensive. It would also yield more complex floorplanning, most likely resulting in a lower operating frequency.

For both versions of the SPS core, a maximum absolute error of 2° is obtained. It can be somewhat difficult to understand the significance of this error. To better understand, the visualization in Figure 7.1 is created, and it is used for visualizing how one degree difference for azimuth is mapped in the two-dimensional plane. This figure visualizes three paths. The reference path is for $\varphi = 180°$, and is marked with a gray, stapled line. The two other paths marks how ±1° error would result in an increasing error for increasing θ.

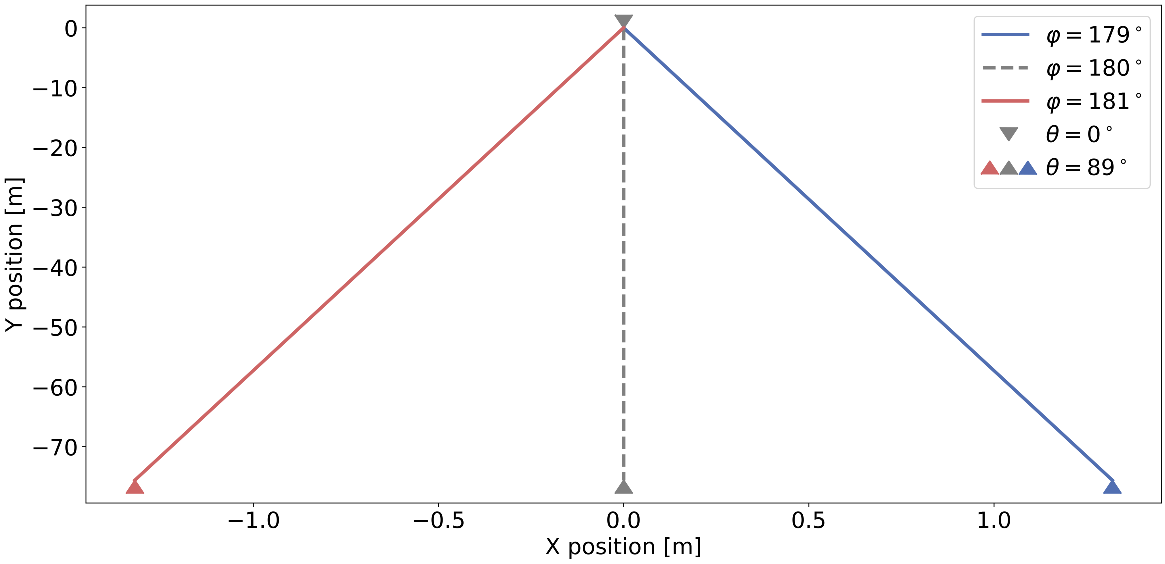


Figure 7.1: Visualization of how ±1° error influence the conversion to a position in 2D. Note that the calculations assume that the height in z-direction is 130 cm between the tag and the locator.

From Figure 7.1, we observe that as $\theta \to 90°$, the error increases. However, by comparing the absolute error to the moved distance, we observe that the error always has a constant ratio compared to the distance from the starting point. It is also observable that the error in y-direction is relatively small, while the error in the x-direction is larger, but still small when comparing the distances in the x-direction to the distances in y-direction. When $\theta = 89°$, we observe that the distance in the y-direction is over 70 meters, and the error in the x-direction is approximately 1.5 meters. With the given system in Section 2.1, this indicates that we will be able to achieve centimeter-level precision, as long as the tag is not moved too far away from the locator in the y-direction.

From Figure 6.14, we observe some errors for φ_E close to 0° for the two-search SPS core. When observing the outputs from these tests, it can be observed that the first search finds $\varphi_{p,1} \in [351°, 360°]$. Remembering the behavior of the two-search SPS core functionality presented in Section 5.3.6, the second search will only be performed on the azimuth values $341° \geq \varphi \geq 360°$, hence missing the actual φ_E. A possible solution for addressing these errors could be to include the azimuth angles $\varphi \in [0°, 9°]$ in the second search for these specific cases. This would yield an additional clock cycle to the two-search SPS core per value of elevation angle. This incorrect behavior is not corrected due to limited time. From Table 6.10, we can observe that the average error is not increased significantly compared to the one-search SPS core, from this incorrect behavior, meaning that it is not critical.

All precision results presented in Section 6.1.1 and Table 6.2 are obtained without any noise.

When working with radio signals, noise is unavoidable, and a discussion of how noise influence the precision of the search should be done. The hardware accelerated SPS is once more tested, this time adding Additive White Gaussian Noise (AWGN) with SNR=30dB to the generated stimuli. The precision plots are shown in Figure 7.2 and Figure 7.3, while a summary of the measured metrics is presented in Table 7.1.

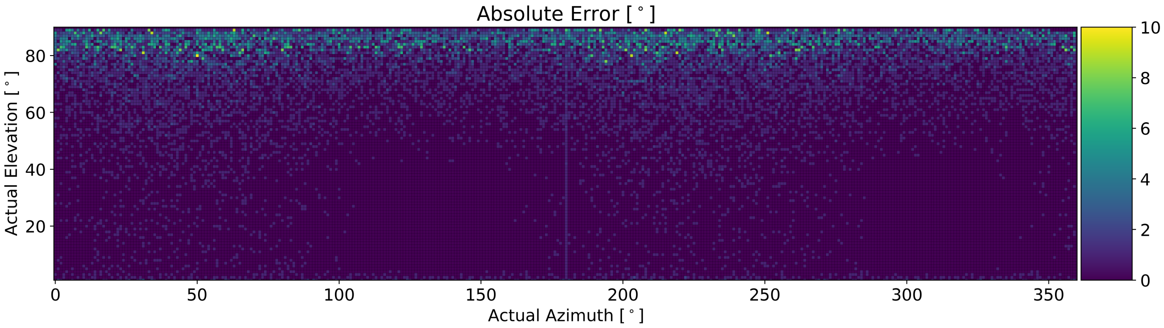


Figure 7.2: One-search SPS core simulation results for SNR=30dB. Waiting for results

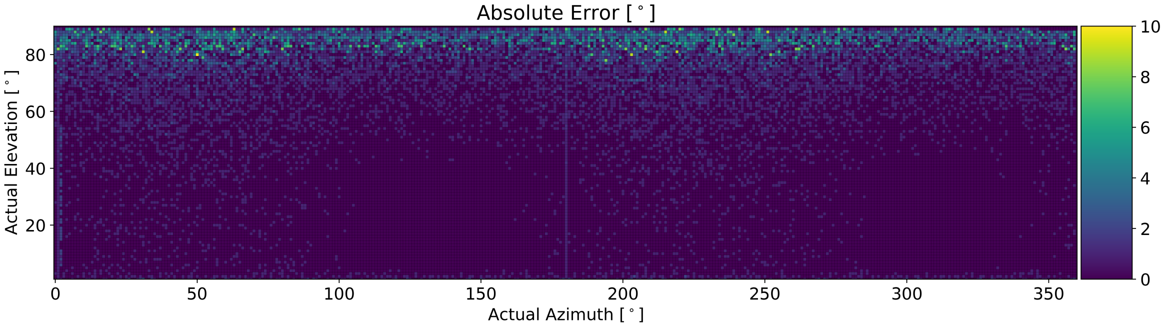


Figure 7.3: Two-search SPS core simulation results for SNR=30dB.

Table 7.1: Precision metrics for the RTL implementation of the SPS core.

Test description	Number of errors	Average absolute error [°]
One-search SPS core	9817	0.38
Two-search SPS core	9912	0.39

From the performed tests, we observe that the number of errors for the two versions is approximately the same, also when noise is present. The difference in number of errors can also be observed in Table 6.10, and it is most likely due to the incorrect behavior of the two-search SPS core for $\varphi_E \approx 0°$.

The added noise clearly increases the number of errors measured for the two versions. However, the magnitude of the majority of the observed errors, particularly for $\theta < 80°$, has an error of 1°. With this in mind, most of these errors could be further reduced or removed with the use of filtering. The performed tests in Figure 7.2 and Figure 7.3 only test every angle combination once, and this is not really comparable to the implemented system. As described in Section 2.1, the tag transmits multiple packets when movement is detected, meaning that if AWGN is present, one would typically get estimates of the AoA centered around the actual AoA with magnitude of the absolute error observed in Figure 7.2 and Figure 7.3. This would imply that the average estimation when receiving multiple packets could be fairly accurate.

7.2 Time and Power Consumption

The timing results obtained for this thesis mainly focus on the implementation of SPS. Although the SPS core is a part of the MUSIC core, accurate timing results are not obtained for the high-level models for the complete algorithm. Opstad developed most of his high-level models in Matlab, making it hard to integrate the implemented SPS high-level models, written in Python.

The obtained timing results for the SPS, both from the high-level model and the implemented hardware versions, are visually presented in Figure 7.4.

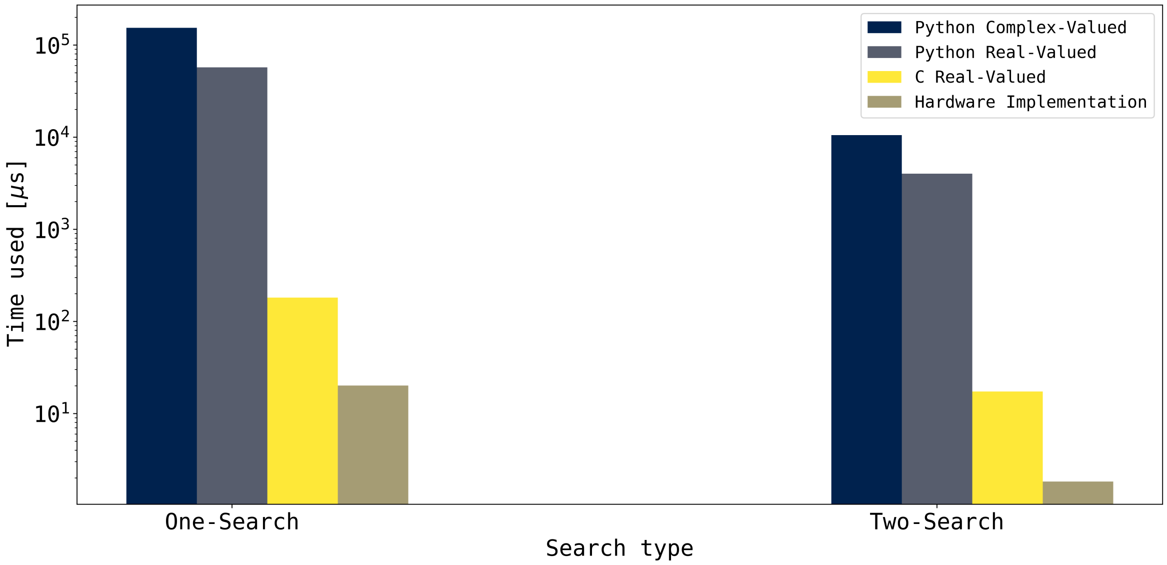


Figure 7.4: Comparison of timing statistics for the performed tests. Not that the y-axis is logarithmic scaled.

From the timing results presented in Figure 7.4, it can easily be observed that the two-search method is generally faster when compared to the one-search method for all versions of SPS. We can also observe that the implemented SPS core achieves the lowest times out of the four versions. However, the C implementation of the SPS scores close to the implemented hardware accelerator. The obtained timing results from the SPS implemented in C can be further explained by observing Figure 6.8. The power increases up to approximately 26W while running the C version of the SPS. The operating frequency of the CPU is significantly higher, allowing more operations to be performed per time unit compared to the hardware accelerated SPS. However, this uses significantly higher power consumption.

With statistics on both required time and power consumption, we can also compare the energy consumption for the high-level models and the implemented design. We know that energy is measured in Joule [J], and it is equal to power [W] · time [s]. For the high-level versions, the difference between the reference power and working power in Figure 6.8 is used for estimating the energy in Table 7.2.

Table 7.2: Energy consumption based on the timing and power analysis. The reported energy consumption is estimated for one full search.

Version	Energy [mJ]
Python CV one-search	2468.3656
Python CV two-search	168.4963
Python RV one-search	917.9064
Python RV two-search	64.3763
C RV one-search	2.9024
C RV two-search	0.2784
HW RV one-search	0.0121
HW RV two-search	0.0013

The analysis of energy consumption indicates that the implemented hardware accelerator outperforms all the other implemented versions. Of course, the power- and energy consumption for the high-level models is rough estimates, as the accuracy of the Open Hardware Monitor application is uncertain. However, the obtained results can be used for indicating that the hardware accelerator is the most energy efficient out of the tested versions.

While discussing the above obtained results, it should also be mentioned that the operating frequency of the hardware accelerator could be adjusted to fit the goals for the system. The above presented results assume that the two versions of the SPS core run at its maximum operating frequency. From the formula for the dynamic power, P_{d}, presented in Equation (7.1), we observe that it highly depends on the frequency, f_{clk}.

$$P_{\mathrm{d}} = \alpha f_{\mathrm{clk}} C_L V_{DD}^2 \tag{7.1}$$

Further analysis of how the hardware accelerator should be used could therefore be performed to find an optimal operating frequency. From the timing results presented in Table 6.14, we observe that the complete MUSIC algorithm takes either 5.34 µs or 24.01 µs, depending on which SPS core version is being used. From Section 2.1, we remember that reading one CTE takes 160 µs, and if the only goal is to complete the algorithm before we can read a new CTE, we could lower the operating frequency significantly, hence reducing P_{d}. However, if one hardware accelerator is to be used for processing the CTE received on multiple locators, the algorithm should be performed with the maximum possible frequency. Further definitions for the use-case of the accelerator must therefore be defined before selecting an appropriate frequency.

An attempt in implementing a minimalistic version of the SPS on the nRF52833 was done to observe the time- and power usage for this SoC. As mentioned in Section 2.1, the nRF52833 is used for sampling the antenna array, and it is located on the selected locator board. If the MUSIC algorithm was to be implemented on this SoC, one would save significant time in transferring the data in- and out of the board. However, when trying to store the steering vector values using 16-bit signed integers on this SoC, overflow occurred in both flash and RAM. This means that the approach of precomputing the steering vector is unachievable, and the values must be computed during run time. A one-search version was implemented, and the execution time for this algorithm was 30 seconds. This indicates that this approach is not preferred, and further attempts in implementing SPS on the nRF52833 are not performed, as we already have compared the hardware accelerated SPS to other versions.

From all the obtained results, it can be observed that the two-search SPS core successfully has reduced the execution time of the search significantly, while maintaining approximately the same level of precision. The previously implemented system, presented in Section 2.1, used a complex-valued MUSIC algorithm. From the obtained timing results, we can observe that the two-search SPS core reduces the search time from 154 ms to 5.34 µs when running at maximum frequency. Some incorrect behavior can be observed in Figure 7.3 and Figure 6.14. However, as discussed above, this is most likely a behavioral error, meaning that it can be solved by adding support for the specific described cases. The two-search SPS core also offers a lower energy consumption per search, resulting in it being the best for the hardware implementation.

7.3 Comparison of Real-Valued and Complex-Valued implementation of SPS Core

When presenting the architecture in the above sections, the focus has been on maximizing the level of parallelism without exceeding the available amount of resources. From Table 6.13, we observe that the presented architecture aims to use approximately 90% of the available DSP slices and 86% of the available BRAM. The architecture for a complex-valued MUSIC algorithm would require a higher utilization percentage if the level of parallelism and bit widths are to be used for storing the steering vectors.

Knowing that a complex-valued number is represented by two real-valued numbers, it can be concluded that using the same number of bits for storing the steering vector values would not be feasible with the chosen SoC and bit-width. When using 18 bits per steering vector value with the chosen BRAM structure in Section 5.3.5, the double amount of BRAM would be required for a complex-values architecture, exceeding the available amount. From the high-level simulations, we have observed that reducing the decimal precision, i.e., reducing the number of bits per value significantly reduced the precision of the implemented search. This indicates that implementing a complex-valued architecture would either be more expensive in terms of memory, or yield a reduction in the precision of the implemented search.

For the DSP slices, we can observe that the implemented `VECMUL` unit would be significantly influenced by a complex-valued architecture. From Equation (3.1), we observe that a complex-valued multiplication requires four real-valued multiplications, which is four times as many as the implemented real-valued architecture. When the real-valued architecture uses 89% of the available DSP slices, it is obvious that it is not feasible to achieve either the same level of parallelism or throughput with complex values. In either case, modifying the architecture to fit complex-valued arithmetic would yield a reduction of performance in terms of speed. Also, increasing the number of resources for supporting the same level of parallelism or using longer time for the search both result in a higher energy consumption, making the difference to the high-level models less.

7.4 Achieving Greater Theoretical Precision

The theoretical minimum error for the implemented design is 0.5°, and it is limited from the decision of using precomputed values for the steering vector. Due to limited available memory, the steering vector is stored with 1° step sizes. However, other FPGA implementations of the MUSIC algorithm, such as [26], achieves a resolution of 0.1°. If the steering vectors were to be stored for this resolution, we would need at least $90/0.1 \cdot 360/0.1 \cdot 12 \cdot 18\text{b} \approx 87.5\text{MB}$, which is significant, and in most designs not feasible as it increases the system cost.

Another way of obtaining higher precision steering vector values is by computing them during run time. This solution is quite computationally heavy, as it would require support for all possible combinations of the trigonometric multiplications, as presented in Equation (4.9). An approximate architecture for computing one steering vector value is presented in Figure 7.5.

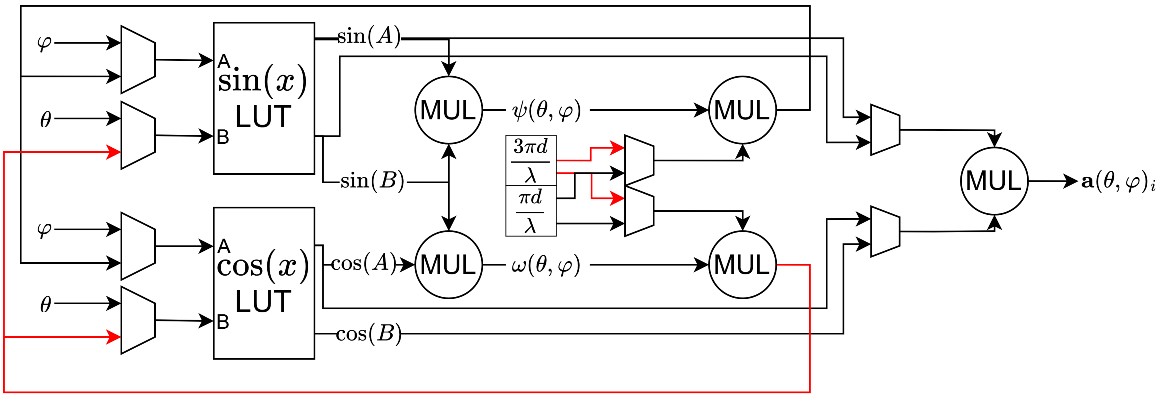


Figure 7.5: Approximate architecture required for computing steering vector values.

LUTs can be used for storing $\cos(x)$ and $\sin(x)$ values with 0.1° steps. Additional multipliers are needed for computing the trigonometric arguments in Equation (4.9) and the multiplications of the cosine and sine expressions. The proposed architecture in Figure 7.5 only computes one steering vector value, and we know that $D = 12$ steering vector values are present for each $\tilde{\mathbf{a}}(\theta,\varphi)$. Adding support for greater precision for the search will therefore require more resources or yield slower execution times for SPS, as we cannot achieve the same level of parallelism when adding support for calculating steering vector values.

7.5 Observations on Non-Constant Wavelengths

As presented in Section 2.2, the transmission of BLE packets can be performed in all the 40 available BLE channels, resulting in the wavelength changing. As seen in Equation (4.9), the wavelength, λ, is used in the steering vector, and it will influence the steering vector values if changed. From Section 5.3.5, the strategy of precalculating all steering vector values and storing them in the available BRAM is presented. This strategy has one drawback, as we are unable to compensate for the changing wavelength when the CTE is received on the different channels.

All performed tests in Chapter 6 have used stimuli that are generated using a constant value for the wavelength. To observe how varying wavelengths influence the precision, a set of simulations is performed using the high-level model. For each of the 40 channels, 20 tests are generated with random AoA using the correct wavelength for the channel. The MUSIC algorithm is performed using the same wavelength for all tests. This is the wavelength used when generating the BRAM data, and it corresponds to channel 17 in Figure 2.4.

The obtained results are presented in Figure 7.6. The performed tests include noise with SNR=30dB and AoA combinations with real-valued numbers, i.e., we must expect errors up to 0.5°, due to the minimal step sizes of 1°.

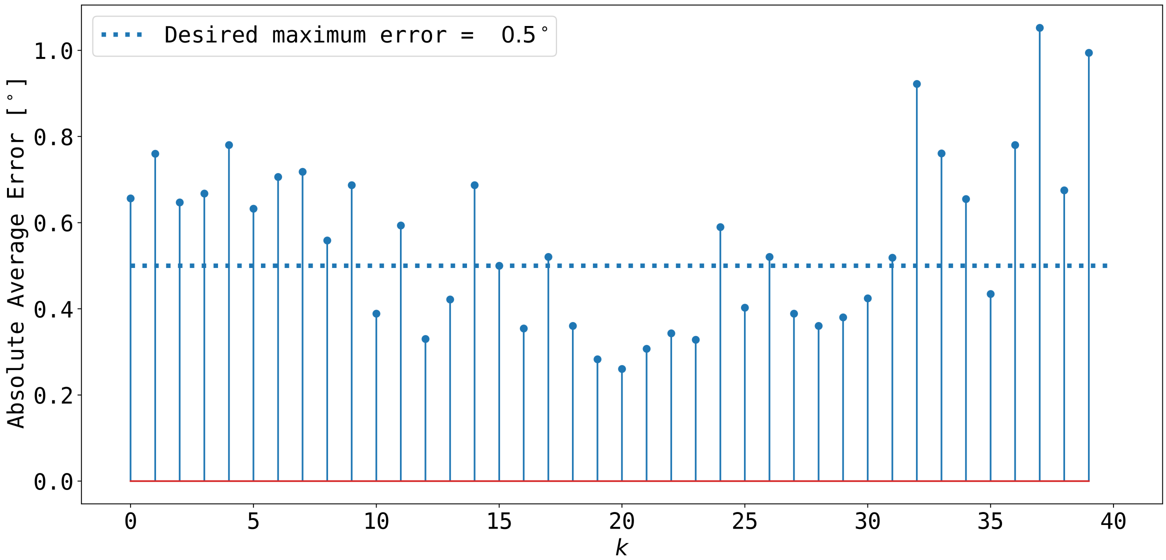


Figure 7.6: Absolute average error for the tests performed with varying λ. Note that $k \in [0, 39]$ describes the tested BLE channel.

The tests clearly indicate that the lowest absolute error is achieved when k is close to 19, which is the middle channel. For the tests with channels close to the edges, we observe a higher absolute average error, meaning that when the CTE is received on these channels, we must expect the AoA estimations to be less precise. However, it is also worth noting that the MUSIC version, developed by EmLogic, in the system discussed in Section 2.1 did not consider the wavelength while executing the algorithm either. This means that it will still be possible to achieve centimeter-level precision with the implemented version of the hardware accelerator. However, for higher level of precision, this should be taken into account. Using the architecture in Figure 7.5 would make it easy to change the wavelength for the different computations.

7.6 Recommendations for Future Work

The below list summarizes the recommendations for future work.

- Test the SPS core with data sampled using the locator board to ensure that the implemented algorithm works in practical applications.

- Integrate the hardware accelerator into existing system. To achieve this, adding support for transferring data in- and out of the FPGA is needed. The MUSIC core can be connected to the Processing System (PS) using Advanced eXtensible Interface (AXI) interface. Moving data from the nRF52833 on the locator board to the PYNQ Z1 board must also be handled.

- Research effective filtering methods for obtaining a more stable behavior for noisy environments.

- Perform more tests with different step size combinations. It is most likely possible to achieve the same level of precision with slightly larger step sizes, reducing the number of iterations further. Observe how large the step sizes can be in noisy environments before the approach becomes unstable.

- Investigate methods for considering the different wavelengths for the available BLE channels.

Chapter 8

Conclusion

As a part of United Nations sustainability goal number 9, RTLS plays a vital role in the industry, allowing for optimized resource management and more effective logistics operations. In 2019, the Bluetooth technology was further improved, allowing the technology to be used in indoor positioning systems with centimeter-level precision. By using Bluetooth Low Energy, a system was implemented to demonstrate the enhancements and capabilities of the technology as a part of the project thesis in the preceding semester. The system consists of a tag that is transmitting Bluetooth Low Energy packets to a receiving antenna array. In the previously implemented system, the received packets are further communicated to a computer, estimating- and visualizing the AoA of the signal. During this project, it was observed that centimeter level precision was achievable, but that the computations on the computer caused significant delay to the system.

In this context, a proposed hardware accelerator is designed for the computations needed to obtain the AoA. The MUSIC algorithm is a popular algorithm for this use case. In this thesis, an application-specific MUSIC algorithm is derived, making optimizations for the ISP1907-AOA-DK antenna layout and for use in Bluetooth Low Energy applications. For reducing the complexity, an additional step of transforming the complex-valued matrices in the MUSIC algorithm to real-valued is performed by using FB averaging and unitary transformation. This transformation allowed for a greater level of parallelism of the implemented hardware accelerator, resulting in faster computation time.

The hardware accelerated algorithm is divided into four steps, and it is implemented by the author of this thesis and Tommy A. Opstad. For the implementation of the hardware accelerator, this thesis has mainly focused on the search needed to estimate the AoA using MUSIC, named the spectral peak search. For this search, multiple architectures and approaches has been presented and discussed. Two versions of the search were implemented: a standard two-dimensional search, named one-search, and a two-search approach. The latter version of the search significantly reduced the iterations of the search, resulting in faster speed of operation while maintaining the same precision as the one-search method. From an analysis of the power and energy consumption, it was observed that the total energy consumption for the two-search version is lower than the standard, one-search version. As the two-search version is almost as precise as the one-search version and more effective, this version is chosen as the preferred method of performing the search in the MUSIC algorithm.

From the complex-valued Python implementation, we observed reduction in time from 154 ms to 5.34 µs, when using the two-search SPS core. When comparing the hardware accelerated search with the high-level models, it is clear that we have been able to reproduce the same behavior, while significantly reducing the power- and time consumption.

The obtained results in this thesis indicate that the implemented hardware accelerator fulfills the goals and objectives presented in the task description and in Section 1.2. The application-specific MUSIC algorithm implemented for the Xilinx XC7Z020 SoC performs better in terms of execution time and energy efficiency while maintaining the same level of precision when compared to software-based, higher levels of implementations.

Bibliography

[1] T. A. Opstad. “FPGA Implementation of a Real-Time Direction Finding System”. Trondheim: NTNU, June 2023.

[2] International Organization for Standardization. *ISO/IEC 24730-1:2014*. ISO. URL: `https://www.iso.org/standard/59801.html` (visited on 05/29/2023).

[3] United Nations. *Goal 9 | Department of Economic and Social Affairs*. URL: `https://sdgs.un.org/goals/goal9` (visited on 05/29/2023).

[4] S. S. Saab and Z. S. Nakad. “A Standalone RFID Indoor Positioning System Using Passive Tags”. In: *IEEE Transactions on Industrial Electronics* 58.5 (May 2011), pp. 1961–1970. ISSN: 1557-9948. DOI: `10.1109/TIE.2010.2055774`.

[5] Bluetooth SIG. *Enhancing Bluetooth Location Services with Direction Finding*. Jan. 25, 2019.

[6] Python Software Foundation. *Welcome to Python.Org*. Python.org. May 29, 2023. URL: `https://www.python.org/` (visited on 05/29/2023).

[7] W. Zhang et al. “Computationally Efficient 2-D DOA Estimation for Uniform Rectangular Arrays”. In: *Multidimensional Systems and Signal Processing* 25.4 (Oct. 2014), pp. 847–857. ISSN: 0923-6082, 1573-0824. DOI: `10.1007/s11045-013-0267-y`.

[8] J. A. Rangnes. “Implementation of a Real Time Locating System Using Bluetooth Low Energy”. Trondheim: NTNU, Dec. 2022.

[9] T. A. Opstad. “Development of Indoor Bluetooth Tracking Tag”. Trondheim: NTNU, Dec. 2022.

[10] Bluetooth SIG. *Bluetooth Core Specification Version 5.1*. Jan. 21, 2019.

[11] D. C. Cassidy, G. Holton, and F. J. Rutherford. *Understanding Physics*. Springer Science & Business Media, Sept. 10, 2002. 857 pp. ISBN: 978-0-387-98756-9.

[12] B. Watson. *FSK: Signals and Demodulation*. Jan. 5, 2001.

[13] Y. Zhao et al. “How to Select the Best Sensors for TDOA and TDOA/AOA Localization?” In: *China Communications* 16.2 (Feb. 2019), pp. 134–145. ISSN: 1673-5447. DOI: `10.12676/j.cc.2019.02.009`.

[14] Digilent. *PYNQ-Z1 Reference Manual - Digilent Reference*. URL: `https://digilent.com/reference/programmable-logic/pynq-z1/reference-manual?redirect=1` (visited on 05/22/2023).

[15] Advanced Micro Devices. *PYNQ - Python Productivity for Zynq*. PYNQ - Python productivity for Zynq. URL: `http://www.pynq.io/` (visited on 06/04/2023).

[16] Xilinx. *Zynq-7000 SoC Data Sheet: Overview V1.11.1*. Feb. 7, 2018.

[17] Xilinx. *7 Series FPGAs Memory Resources User Guide (UG473)*. Mar. 7, 2019.

[18] Xilinx. *7 Series DSP48E1 Slice User Guide (UG479)*. Mar. 27, 2018.

[19] R. O. Schmidt. “Multiple Emitter Location and Signal Parameter Estimation”. In: *IEEE TRANSACTIONS ON ANTENNAS AND PROPAGATION.* 3rd ser. AP-34 (Mar. 1986), pp. 276–280.

[20] Z. Zou, W. Hongyuan, and Y. Guowen. “An Improved MUSIC Algorithm Implemented with High-speed Parallel Optimization for FPGA”. In: *2006 7th International Symposium on Antennas, Propagation & EM Theory.* 2006 7th International Symposium on Antennas, Propagation & EM Theory. Oct. 2006, pp. 1–4. DOI: 10.1109/ISAPE.2006.353475.

[21] P. Gupta and S. Kar. “MUSIC and Improved MUSIC Algorithm to Estimate Direction of Arrival”. In: *2015 International Conference on Communications and Signal Processing (ICCSP).* 2015 International Conference on Communications and Signal Processing (ICCSP). Apr. 2015, pp. 0757–0761. DOI: 10.1109/ICCSP.2015.7322593.

[22] X.-T. Meng et al. “Real-Valued MUSIC for Efficient Direction of Arrival Estimation With Arbitrary Arrays: Mirror Suppression and Resolution Improvement”. In: *Signal Processing* 202 (Jan. 2023), p. 108766. ISSN: 01651684. DOI: 10.1016/j.sigpro.2022.108766. URL: https://linkinghub.elsevier.com/retrieve/pii/S016516842200305X (visited on 03/08/2023).

[23] J. Cai et al. “A Derivative-Based MUSIC Algorithm for Two-Dimensional Angle Estimation Employing an L-Shaped Array”. In: *2020 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT).* 2020 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT). Dec. 2020, pp. 1–5. DOI: 10.1109/ISSPIT51521.2020.9408790.

[24] Keh-Chiarng Huarng and Chien-Chung Yeh. “A Unitary Transformation Method for Angle-of-Arrival Estimation”. In: *IEEE Transactions on Signal Processing* 39.4 (Apr. 1991), pp. 975–977. ISSN: 1053587X. DOI: 10.1109/78.80927.

[25] W. Si et al. “Real-Valued 2D MUSIC Algorithm Based on Modified Forward/Backward Averaging Using an Arbitrary Centrosymmetric Polarization Sensitive Array”. In: *Sensors* 17.10 (Sept. 29, 2017), p. 2241. ISSN: 1424-8220. DOI: 10.3390/s17102241.

[26] K. Huang et al. “An Efficient FPGA Implementation for 2-D MUSIC Algorithm”. In: *Circuits, Systems, and Signal Processing* 35.5 (May 2016), pp. 1795–1805. ISSN: 0278-081X, 1531-5878. DOI: 10.1007/s00034-015-0144-z. URL: http://link.springer.com/10.1007/s00034-015-0144-z (visited on 01/19/2023).

[27] A. Nicolaides. *Pure Mathematics: Complex Numbers.* Pass Publications, 2007. 97 pp. ISBN: 978-1-872684-92-5. Google Books: jysDhWH4CKEC.

[28] “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (July 2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[29] R. A. Horn and C. A. Johnson. *Matrix Analysis.* Cambridge: Cambridge University Press, 1985. ISBN: 0-521-30586-1.

[30] G. H. Golub and C. F. Van Loan. *Matrix Computations.* 3rd ed. Baltimore, Maryland: The Johns Hopkins University Press, 1996. ISBN: 0-8018-5413-X.

[31] International Telecommunication Union. *Data Format Definition for Exchanging Stored I/Q Data for the Purpose of Spectrum Monitoring.* Sept. 2018.

[32] F.-G. Yan et al. “Two-Dimensional Direction-of-Arrivals Estimation Based on One-Dimensional Search Using Rank Deficiency Principle”. In: *International Journal of Antennas and Propagation* 2015 (2015), pp. 1–8. ISSN: 1687-5869, 1687-5877. DOI: 10.1155/2015/127621.

[33] *Estimation of Covariance Matrices.* In: *Wikipedia.* Jan. 11, 2023. URL: https://en.wikipedia.org/w/index.php?title=Estimation_of_covariance_matrices&oldid=1132954415#Intrinsic_covariance_matrix_estimation (visited on 03/13/2023).

[34] D. Linebarger, R. DeGroat, and E. Dowling. “Efficient Direction-Finding Methods Employing Forward/Backward Averaging”. In: *IEEE Transactions on Signal Processing* 42.8 (Aug. 1994), pp. 2136–2145. ISSN: 1941-0476. DOI: `10.1109/78.301848`.

[35] Insight SIP. *Application Note AN210401*. 2021.

[36] Q. Spencer et al. “Indoor Wideband Time/Angle of Arrival Multipath Propagation Results”. In: *1997 IEEE 47th Vehicular Technology Conference. Technology in Motion.* 1997 IEEE 47th Vehicular Technology Conference. Technology in Motion. Vol. 3. Phoenix, AZ, USA: IEEE, 1997, pp. 1410–1414. ISBN: 978-0-7803-3659-9. DOI: `10.1109/VETEC.1997.605455`. URL: `http://ieeexplore.ieee.org/document/605455/` (visited on 06/06/2023).

[37] *Built-in Functions — Python 3.10.11 Documentation*. URL: `https://docs.python.org/3.10/library/functions.html#round` (visited on 05/23/2023).

[38] M. Moller. *Open Hardware Monitor*. Version 0.9.6. Dec. 27, 2020. URL: `https://openhardwaremonitor.org/`.

[39] NumPy community. *NumPy User Guide*. June 22, 2022. URL: `https://numpy.org/doc/1.23/numpy-user.pdf`.

Appendix A

Vivado Power Report Default Parameters

Table A.1: Vivado default power report parameters.

Parameter	Default Value
Ambient Temperature	25°C
Board Temperature	25°C
Airflow	250 LFM
Heat Sink	None
Output Load	0 pF

NTNU

Norwegian University of
Science and Technology

EmLogic