Simen Hansen Skåra
Dawid Litwicki
Trym Brabrand

Bachelor's thesis

Location of obecjt with help of sound triangulation

May 2023

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

NTNU
Norwegian University of
Science and Technology

Bachelor's thesis

2023

NTNU
Norwegian University of
Science and Technology

Simen Hansen Skåra
Dawid Litwicki
Trym Brabrand

Location of obecjt with help of sound triangulation

Bachelor's thesis
May 2023

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

NTNU
Norwegian University of
Science and Technology

ABSTRACT

Kongsberg Maritime has expressed interest in using sound to automate boat location tracking. The bachelor group is entrusted with doing research into efficient ways to carry out this.

The thesis will simulate a boat that send some form of sound signals. Several microphones and an FPGA (Field-Programmable Gate Array) will be used to sample the signals. Then, using signal processing, determine how long the sound waves' arrival delay is. After that, calculate the object's direction and distance and display the results on a GUI (Graphical User Interface).

The bachelor group was unable to finish the prototype due to a lack of time. The FPGA took longer than expected to complete. As a result, the group was unable to fine-tune the inputs and alter the variables in the various classes.

CONTENTS

LIST OF FIGURES

ABBREVIATIONS

List of all abbreviations in alphabetic order:

- **ADC** Analog to digital converter
- **API** Application programming interface
- **AXI** Advanced Extensible Interface
- **FFT** Fast Fourier Transformation
- **FPGA** Field-programmable gate array
- **GUI** Graphical user interface
- **HDL** Hardware descriptive language
- **IPv4** Internet protocol version 4
- **ISR** Interrupt Service Routine
- **lwIP** Ligthweight IP
- **MAC** Media access control
- **OS** Operating System
- **PCB** Protocol control block
- **RAM** Rapid access memory
- **SNR** Signal to Noise Ratio
- **UART** Universal asynchronous receiver-transmitter
- **UDP** User diagram protocol4
- **wav** Waveform Audio File Format

CHAPTER

ONE

INTRODUCTION

1.1 Background

According to maritime regulations, a boat must be able to communicate through sound signals of a foghorn[1]. This task should be automated, in accordance with Kongsberg Maritime. In addition, Kongsberg Maritime tasked the group to investigate the reliability of location estimation of a sound source, and understand the issues that could appear in the signal processing.

1.2 Objective

The main objective of the project is to automate sound source location between two ships by the use of sound triangulation, in an simulated environment. The pipeline evolves around receiving the sound by an FPGA, signal processing the data in the frequency domain if the desired frequency is detected, extracting the timestamps from the microphones, and finally estimating the location of the source in the form of angle and distance estimators.

The objectives for this thesis are:

1. To create a prototype with the ability to efficiently sample a signal that simulates a foghorn.
2. Process the recorded signals when the desired frequency is detected, and extract time samples from the microphone readings.
3. Use the time stamps as a basis for angle and distance estimations.
4. Present the estimated position on a GUI.
5. Examine the various methods that were developed and talk about their viability.

1.3 Structure of the rapport

- **Chapter 2 - Theory** - Introduction to the theoretical background which is the basis of this thesis.
- **Chapter 3 - Methods** - Contains a description of the methodology and materials that were considered throughout the project.
- **Chapter 4 - Results** - Showcase of what the group accomplished throughout the project.
- **Chapter 5 - Discussion** - Discussion of the project's planning, group process, outcomes, benefits and drawbacks of the prototype, and recommendations for further improvement.
- **Chapter 6 - Conclusions** - Presentation of overall conclusion and final results of the whole assignments.

CHAPTER

TWO

THEORY

2.1 Hardware

2.1.1 Raspberry pi

Raspberry Pi is a UK tech company that manufacture micro computers that can run multiple operating systems (OS), making them versatile[2]. Raspberry Pi has developed their own Linux-based OS with python[3] as a central part of the built in application[2].

2.1.2 Field Programmable Gate Array

A field programmable gate array, FPGA for short, is a chip that consist of configurable logic blocks. They are specialized for performing high speed applications like signal processing, low latency networking and so on[4]. The truth tables are connected so that they can be programmed to do close to anything[5].

An FPGA's manufacturer can deliver FPGA's with pre-programmed parts called "hard-cores"[5]. These "hard-cores" are usually widely used in digital systems[5], like this thesis project. If the pre-programmed parts have programmable functions or can be configured they are called "soft-cores"[5]. FPGAs with either "hard- or soft-cores" or both are called platform FPGAs, they can implement whole systems in themselves[5]. The manufacturer Xilinx has called "hard- and soft-cores" for "IP-cores" and they will be used for programming the FPGA.

2.1.3 Analog to Digital Converter

An analog to digital converter (ADC) is used to replicate an analog signal as a digital one. It does this by measuring a voltage or current that is proportional with the analog signal with a given frequency. The greater the voltage or current is the greater the returned bit-value is[6].

As figure 2.1.1 illustrates, the Arty has two 12-bit internal ADCs that can operate at a speed of one mega sample per second[7].

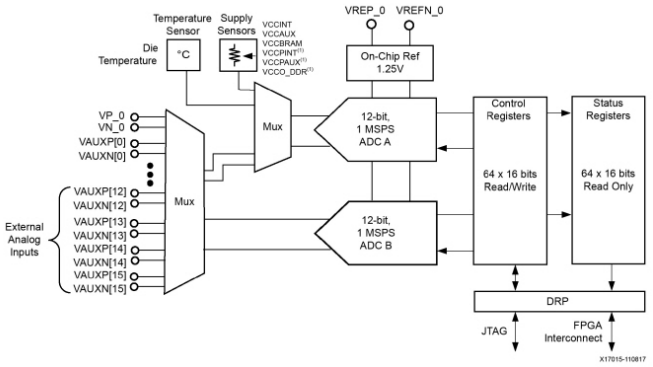


Figure 2.1.1: XADC module overview[7].

When the ADC is not used in it's differential mode it can only measure between zero and one volt making the output wave fluctuate around 0.5 volt instead of zero. This offset is the main reason for DC-bias to occur[6].

The ADC is instantiated and configured by the IP-core[7]. The hardware could be setup to be sampling continuously. When configured to do so, the ADC can transfer data to the microprocessor through AXI4 protocol[7]. When AXI4 is enabled the ADC can be configured by software through AXI channels. It can be configured to operate in interrupt mode or polling mode as well.

Polling mode is when the software constantly checks if the ADC has ready data. Interrupt mode on the other hand sets up an interrupt controller to let the microprocessor know when data is ready[7]. That way the program can perform other tasks while waiting for the interrupt to be called. The microprocessor can be configured to perform a certain function every time the interrupt controller gives the signal.

Equation 2.1 finds the time it takes to fill a number of buffers in seconds.

$$time\_to\_fill\_x\_buffers = \frac{buffer\_size \cdot number\_of\_buffers}{sampling\_rate} \tag{2.1}$$

2.2 Communication System

2.2.1 Advanced Extensible Interface protocol

The AXI protocol supports high-performance, high-frequency system designs[8]. It is an on-chip master and slave communication protocol[8]. On Xilinx's FPGAs it is widely used as the communication protocol between the IP-cores.

2.2.2 Cross platform communication

For one platform to communicate with another one multiple protocols are at play. The internet protocol suite is a term that gathers all the protocols that work together to transfer data cross platform[9]. Four layers make up the internet protocol suite[9]. The four layers are; the link layer, internet layer, transport layer and application layer[9].

The library "Lightweight IP" (lwIP) is an open source library designed for embedded systems[10]. It follows a standard that combines the layers, and allows a user to make configure the network interface and make callback functions to connect the users application to the library. When programming on an embedded system with no OS the raw API has to be used[10]. It require more setup from the user. LwIP combines parts of the link layer and internet layer into a C[11] object called "netif", short for network interface. The netif object stores the MAC-address of the hardware and the start- and end point IP(v4) addresses.

User datagram protocol (UDP) is a transport layer protocol that excels on speed and broad-casting[12]. UDP's speed is great because the protocol does not present a handshake that guarantee that the package reached it's destination.

To make a link between two or more computers a protocol has to begin the communication and establish the link. The transport layer nor internet layer has an internal routine for making that link. Therefore protocols on the link layer like "Address resolution protocol" (ARP) is necessary for network communication[13].

ARP discovers the link layer's address, like the MAC-address[13].

To hold the various information that the UDP needs a protocol control block is needed (PCB). A PCB holds the information common for all transport layer end points[14].

2.3 Signal processing

2.3.1 DC Removal

The output of an analog-to-digital converter often has a tiny DC bias when digitizing analog signals, meaning that the average of the digitized time sample is not zero. An audio signal with a nonzero DC bias level is particularly problematic since switching between audio signals or concatenating two audio signals causes annoying clicks.

DC elimination is simple if the signal is processed non-real time and the signal data is acquired in blocks of block length N. To create a new time sequence with a very low DC bias, simply calculate the average of N time samples and subtract that average value from each original sample [15].

2.3.2 Harmonic Product Spectrum

HPS measures the maximum coincidence for harmonics for each spectral frame according to the following equation 2.2 [16].

$$Y(\omega) = \prod_{r=1}^{R} |X(\omega r)| \tag{2.2}$$

The resulting periodic correlation array $Y(\omega)$ can then be searched for the highest value [16]. A visualisation of the equations in use can be seen in figure 2.3.1.

$$\hat{Y} = \max_{\omega_i} Y(\omega_i) \tag{2.3}$$

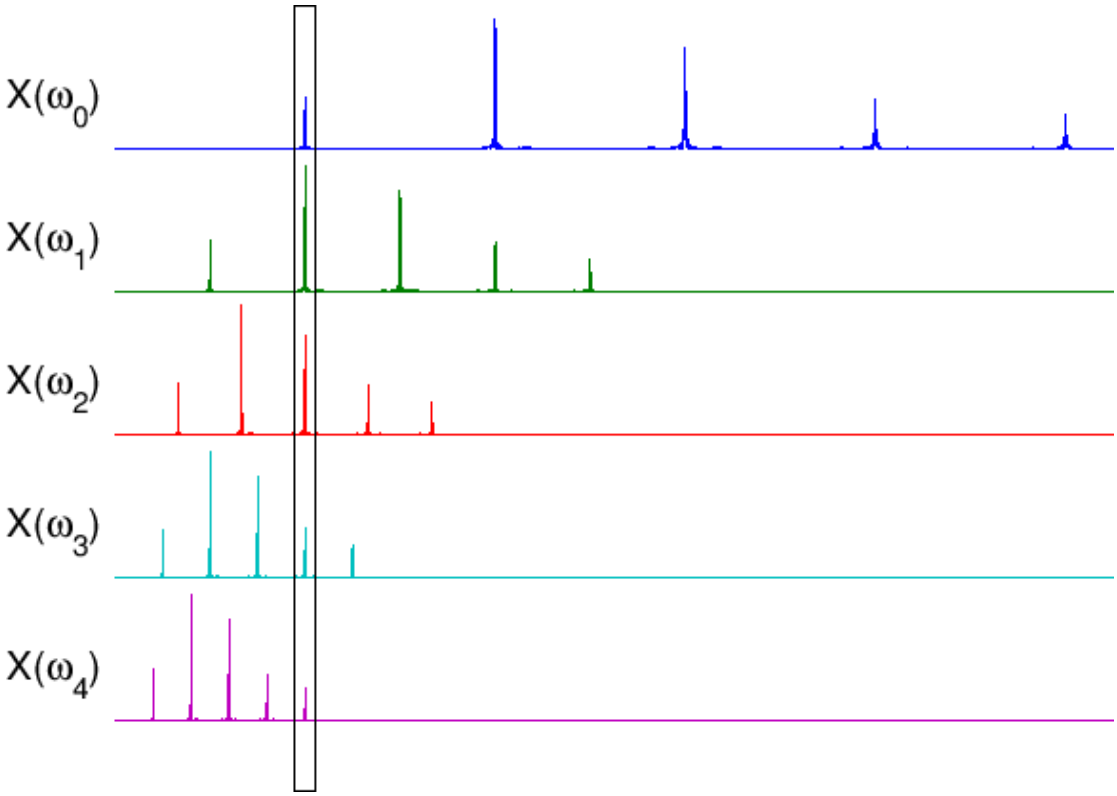


Figure 2.3.1: Example of Harmonic product spectrum

2.3.3 Cross correlation

Cross-correlation is convolution of two signals where the second signal is conjugated. Convolution is a mathematical way of combining two signals to form a third signal. The third signal represents the similarity between the two signals 2.4.

$$y[n] = \sum_{k=-\infty}^{\infty} x_1[k] \cdot x_2[n-k] \tag{2.4}$$

This equation can be transformed into the frequency domain, where it can be performed by multiplying the two signals, with the second signal being conjugated 2.5.

$$y = x_1 * {x_2}^* \tag{2.5}$$

2.3.3.1 Noise whitening

Whitening allows one to only take phase of $X_i(k)$ into account, giving each frequency component the same weight and narrowing the wide maxima caused by correlations within the received signal. This is used in order to narrow down the peak of cross-correlation [17][18].

2.3.3.2 Spectral weighing

In order to detect specific sounds, one can take signal to noise ratio (SNR) into account. This can be done be weighing the different frequencies with the following equation [18].

$$w(k) = \max(0.1, \frac{X(k) - \alpha X_n(k)}{X(k)}) \tag{2.6}$$

where $\alpha < 1$ is a coefficient that makes the noise estimate more conservative. In order to increase the contribution of tonal regions of the spectrum (where the local SNR is very high), one can define enhanced weighting function $w_e(k)$ as [18]:

$$we(k) = \left\{ \begin{array}{ll} w(k) & , X(k) \leq X_n(k) \\ w(k)(\frac{X(k)}{X_n(k)})^\gamma & , X(k) > X_n(k) \end{array} \right\} \tag{2.7}$$

2.4 Angle and direction estimation

2.4.1 Angle estimation

The thesis uses two alternative methods to calculate angles. The first technique calculates several angles using the various times between the microphones and the distance between the same microphones. The second method calculates a single angle using vector computations, but both approaches begin by assuming that sound waves move in a straight line rather than a circle.

In order to calculate distance, both methods first multiply the sound speed by the time difference. It is feasible to create a triangle with this distance as shown in figure 2.4.1

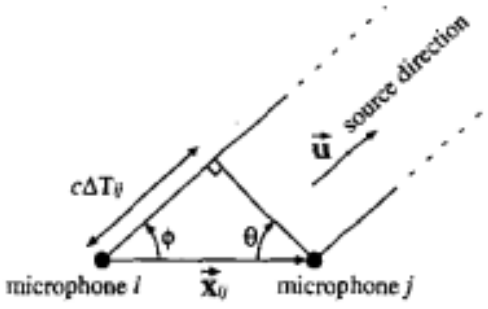


Figure 2.4.1: Computing source direction from TDOA [18]

Here, the various approaches start to diverge. The first technique uses the equation 2.8 to calculate the angles between each time difference and obtains

more angles with additional microphones. In actuality, this approach determines a triangle's angle. As a result, each angle has a unique offset that corresponds to the microphone's initial position; it is crucial to offset them in order for the angle to represent the correct coordinate system.

$$angle_{ij} = arccos(\frac{c \cdot \Delta T_{ij}}{\sqrt{(|x_i| + |x_j|)^2 + (|y_i| + |y_j|)^2}}) + offset \tag{2.8}$$

The second method, however, calculates the distance based on the time difference using a vector method, as illustrated in equation 2.9 [18].

$$\overrightarrow{u} \cdot \overrightarrow{x_{ij}} = c\Delta T_{ij} \tag{2.9}$$

Where $\overrightarrow{u}$ is a unit vector pointing in the source's direction, $\overrightarrow{x_ij}$ is a vector connecting microphones i and j, and c is the speed of sound. You might rewrite this as shown in equation 2.10 [18].

$$u(x_j - x_i) + v(x_j - x_i) + w(z_j - z_i) = c\Delta T_{ij} \tag{2.10}$$

Equation 2.11 illustrates how, given N microphones, one might arrive to a set of N-1 equations [18].

$$\begin{bmatrix} (x_2 - x_1) & (y_2 - y_1) & (z_2 - z_1) \\ (x_3 - x_1) & (y_3 - y_1) & (z_2 - z_1) \\ \vdots & \vdots & \vdots \\ (x_N - x_1) & (y_N - y_1) & (z_2 - z_1) \end{bmatrix} \cdot \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} c\Delta T_{12} \\ c\Delta T_{13} \\ \vdots \\ c\Delta T_{1N} \end{bmatrix} \tag{2.11}$$

Because the angle to the object is contained in the unit vector, it is possible to find the unit vector and angle to the object by applying pseudo-inverse to the $\overrightarrow{x_ij}$ vector [18].

It is possible to obtain an angle from a sound-producing object with some margin of error using these two techniques. Look through [18] for more details about the second method.

2.4.2 Distance estimation

Utilizing the angles determined in the angle estimation section 2.4.1 is necessary to determine the object's direction. Polar coordinates must be converted to Cartesian coordinates in order to find the distance. For this, the equation 2.12 was used.

$$\begin{aligned} x &= r \cdot cos(\theta) \\ y &= r \cdot sin(\theta) \end{aligned} \tag{2.12}$$

With the various coordinates, lines may now be created that connect to their own starting locations. Now there is an intersection of several lines going in the same direction. The object is located at the junction, with some possibilities of margin.

CHAPTER

THREE

METHODS

The terminal experiment that provides all the results for the live test follows these conditions:

A computer was used to play a piano note with the fundamental frequency of 440Hz one meter from the prototype. The prototype is oriented with a zero degree angle from the sound source. This test will be used as a basis for the live data, in the result section of this rapport.

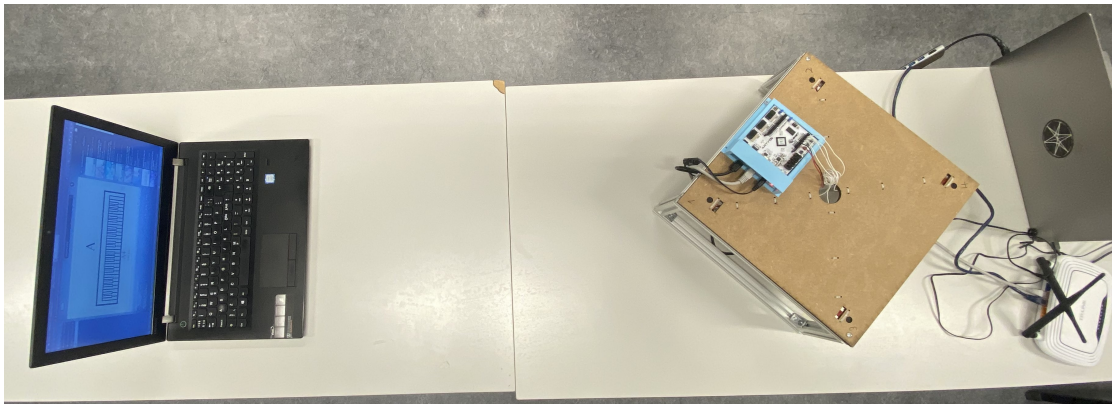


Figure 3.0.1: Experiment setup

Multiple smaller experiments were conducted under the same conditions to further gather- and analyze data- and result.

3.1 Hardware

3.1.1 Equipment

- Arty A7 35T FPGA
- SparkFun Electret Microphone Breakout
- Platform

3.1.2 Design

It was decided to arrange the four microphones in a square arrangement for this thesis. There is a ideal distance to maintain sufficient wave tops when there is cross-correlation between sound sources. This formula can be used to explain this distance 3.1.

$$dist_M = \frac{c}{f_M \cdot 2} \tag{3.1}$$

c denotes the speed of sound, which for the purposes of this project is estimated to be 343 m/s. f_M represents the highest frequency the system can detect, and $dist_M$ the greatest distance between the microphones. The group determined that the prototype could detect frequencies up to 440 Hz, therefore using the equation 3.1, they were able to determine that the maximum distance between the microphones should be 38.98 cm. Pythagoras formula [19] can be used to calculate the distance between square sides. This distance model was created with this data.

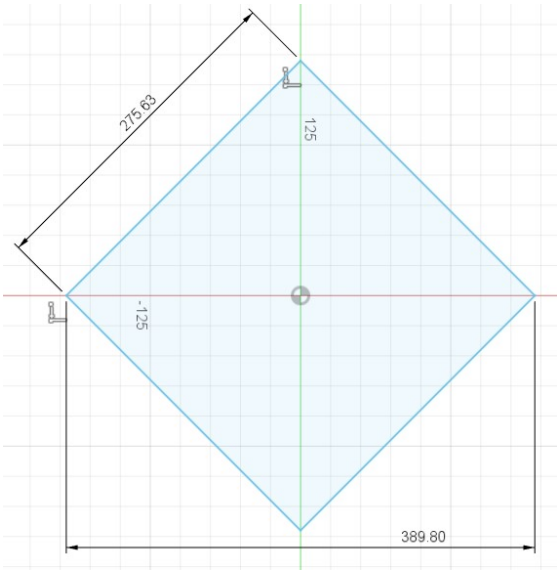


Figure 3.1.1: Design of microphone placement with measurement in mm

3.1.3 Platform

A holding device was created for the platform in order to mount the FPGA to it. Easy installation and dismounting were prioritized throughout the holding unit's design process because the FPGA was still in development at the time.

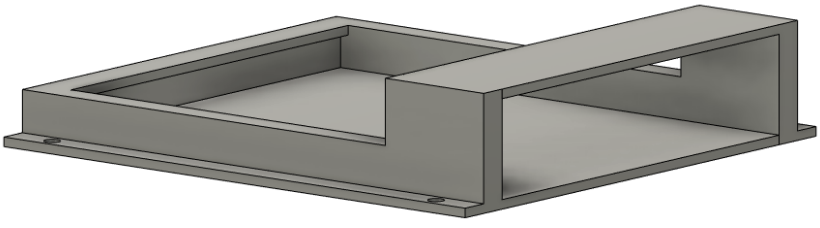


Figure 3.1.2: CAD file of the unit holder

The finishing platform is 34 cm by 34 cm by 10 cm. It was decided to raise the platform 10 cm above the ground to make it easier to move around. The platform can been seen in figure 3.1.3.

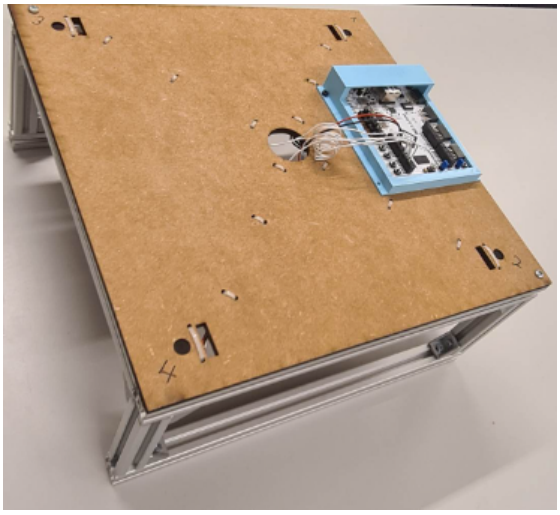


Figure 3.1.3: Picture of platform

3.1.4 Field Programmable Gate Array

So Google was used as a search engine to look into what an FPGA even was and find a suitable one for the application. On "Diligent.com" a suitable FPGA platform with an on-chip ADC and microprocessor was found[20]. Xilinx AMD is the vendor of the chosen FPGA; the Arty A7 35t evaluation board.

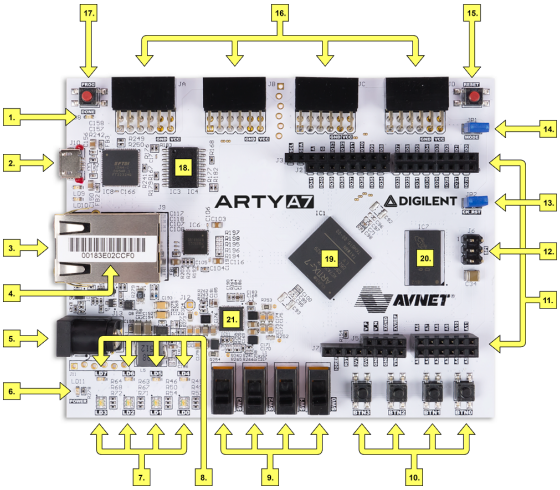


Figure 3.1.4: Arty A7 evaluation board [20].

3.1.4.1 Vivado suite

The Vivado suite is the collective software tool for all Xilinx's FPGA's and evaluation boards. It has configurable IP-cores to place and interconnect to make a hardware design with desirable characteristics.

The IP's needed for the application of sampling an ADC and transfer data with ethernet is:

- System clock
- DDR3 memory
- Microblaze processor
- AXI slave- master interconnect
- AXI timer
- Interrupt controller
- System- and processor reset
- XADC wizard
- UART
- Ethernetlite

A good base is using the Microblaze base project in Xilinx's tutorials[21]. From there add the remaining IP's and let the DDR3 ui clock drive the entire system. Configure everything to communicate on the AXI4 protocol, and lastly reconfigure the DDR3 to not create an instance of the XADC to monitor the internal temperature.

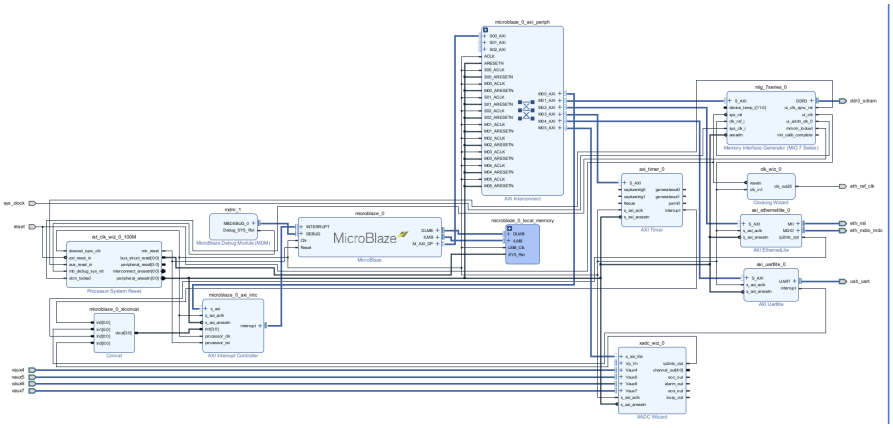


Figure 3.1.5: Hardware diagram of FPGA platform. TODO (made by me)

3.1.4.2 Vitis SDK

If designed correctly Vivado should be able to generate a HDL wrapper and bitstream from it. From there export the hardware with the "include bitstream" box checked. The exported hardware file is needed to create a platform project in the Vitis SDK. On the platform all the hardware drivers are located. Additionally a library for internet protocols and configuration is needed. Lightweight IP can be located on the "board support package" tab in Vitis.

Lightweight IP's source file "xadapters.c" has a syntax error that needs to be fixed in order for the library to work properly. Inside the function "xemac_add" there is a switch-case which is missing break statements for each case, which makes this switch always resort to the default state that returns an error message. The Vitis template "TCP echo server" was the base for the application layer for the previously made platform. To refactor the echo server to use UDP instead of TCP a Github repository was used as a guide[22]. The callback function in figure 3.1.6 for handling received messages shouldn't be needed, but for stability of the application it was kept and will simply echo back the received message.

```
void udp_echo_recv(void *arg, struct udp_pcb *pcb_p, struct pbuf *p, ip_addr_t *ip, u16_t port)
{
    if (p != NULL) {
        /* send received packet back to sender */
        udp_sendto(pcb_p, p, ip, port);
        /* free the pbuf */
        pbuf_free(p);
    }
}
```

Figure 3.1.6: UDP receive message callback function[22].

Static IP addresses were used for the project. The default IP address of the board was "192.168.1.10", but it could be changed. The IP address used for the receiving end was "192.168.1.50", it could also be changed. The netmask was set to the default value. Finally, the boards MAC address is found on the sticker of the ethernet port on the arty board as illustrated by arrow four in figure 3.1.4. It is a string of hexadecimal numbers. All of this was passed to a netif function to setup the network interface.

```
int start_application(const ip_addr_t *local_ip, const ip_addr_t *remote_ip)
{
    err_t err;

    /* create new TCP PCB structure */
    pcb_p = udp_new();
    if (!pcb_p) {
        xil_printf("Error creating PCB. Out of Memory\n\r");
        return -1;
    }

    /* bind to specified @port */
    err = udp_bind(pcb_p, local_ip, REMOTE_UDP_PORT);
    if (err != ERR_OK) {
        xil_printf("Unable to bind to port %d: err = %d\n\r", REMOTE_UDP_PORT, err);
        return -2;
    }

    udp_recv(pcb_p, (udp_recv_fn) udp_echo_recv, NULL);

    //udp_connect(struct udp_pcb *pcb_p, const ip_addr_t *ipaddr, u16_t port)
    err = udp_connect(pcb_p, remote_ip, REMOTE_UDP_PORT);
    if (err != ERR_OK)
    {
        xil_printf("Unable to connect: err = %d\n\r", err);
        return -2;
    }


    xil_printf("UDP echo server started @ port %d\n\r", REMOTE_UDP_PORT);

    return 0;
}
```

Figure 3.1.7: Start application function.

The function in figure 3.1.7 sets up the PCB with receive callback function and end point's information. The PCB is instantiated by the function "udp_new" and filled by the functions; "udp_bind", "udp_connect" and "udp_recv". Bind inserts the local end-point information in the PCB while connect inserts the remote

end-point information, and recv places a pointer to the callback function in the PCB.

To interact with the ADC through C the driver xSysMon is presented by Vitis. The driver needs to be configured for interrupt driven mode to be able to receive samples at the desired frequency. Additionally, to read from the ADC channels, bit-masks are defined in the xSysMon header files. Make sure to not confuse the masks to enable the channels for the masks to read them. The bit-masks are **not** equal!

xSysMon does not provide a Interrupt Service Routine (ISR). To make one, a callback function has to be made which describes what to do when the interrupt has been issued. A pointer to the callback function and the XSysMon-instance is then passed to the instance of the interrupt controller to connect them. XSysMon presents a function to enable interrupts for certain interrupt-registers. The driver also has bit-masks to use to identify specific interrupt-registers on the physical ADC.

```
/*****************************************************/
void StartAdc()
{
    /*
     * Enable end of sequence interrupt for vaux channels.
     */
    XSysMon_IntrEnable(SysMonInstPtr, XSM_IPIXR_EOS_MASK);

    /*
     * Enable global interrupt of System Monitor.
     */
    XSysMon_IntrGlobalEnable(SysMonInstPtr);
}
```

Figure 3.1.8: Start ADC function.

Because the enabled interrupt was for the "end of sequence" register the optimal place to get the values from the ADC was inside the interrupt callback function. The "end of sequence" register goes high each time the ADC has completed a conversion for each channel that is enabled and the data is ready to be read. The data is transferred directly into a big buffer that is allocated in the RAM. The transfer is done by creating a pointer which points to an address in the buffer. The data is then redirected to the memory where the pointer is pointing, and the pointer is incremented by one. Effectively meaning that the pointer points to the next address in the memory. This is how the data is gathered in the FPGA, observe figure 3.1.9b for source code.

```c
/*****************************************************/
int SysMonSetupInterruptSystem(u16 IntrId) {

    int Status;
    XIntc *intcp;
    intcp = &intc;

    /*
     * Connect the handler that will be called when an interrupt
     * for the device occurs, the handler defined above performs the
     * specific interrupt processing for the device.
     */
    Status = XIntc_Connect(intcp,
                IntrId,
                (XInterruptHandler) SysMonInterruptHandler,
                SysMonInstPtr);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /*
     * Enable the interrupt for the System Monitor/ADC device.
     */
    XIntc_Enable(intcp, IntrId);

    return XST_SUCCESS;
}
```

(a) ADC's interrupt routine setup.

```c
/*****************************************************
                ADC interrupt handler:
*****************************************************/
void SysMonInterruptHandler(void *CallBackRef)
{
    u32 IntrStatusValue;
    XSysMon *SysMonPtr = (XSysMon *) CallBackRef;

    IntrStatusValue = XSysMon_IntrGetStatus(SysMonPtr);  // Get Interrupt Status Register

    if (IntrStatusValue & XSM_IPIXR_EOS_MASK)  // If this is End Of Sequence interrupt-
    {
        if (adc_buff_write_enable)  // -and we are allowed to write into adc_buffer:
        {
            // Move samples from the ADC into adc_buffer:
            *(adc_buff_write_p++) = XSysMon_GetAdcData(SysMonInstPtr, XSM_CH_AUX_MIN+4);
            *(adc_buff_write_p++) = XSysMon_GetAdcData(SysMonInstPtr, XSM_CH_AUX_MIN+5);
            *(adc_buff_write_p++) = XSysMon_GetAdcData(SysMonInstPtr, XSM_CH_AUX_MIN+6);
            *(adc_buff_write_p++) = XSysMon_GetAdcData(SysMonInstPtr, XSM_CH_AUX_MIN+7);

            if (adc_buff_write_p >= &adc_buffer[2*ADC_BUFFER_SIZE])  // We are at the end of buffer-2:
            {
                adc_buff_write_p = adc_buffer;  // Start at beginning of buffer-1 next time
                adc_buff_2_filled = TRUE;       // Tell main-loop that buffer-2 is ready
                if (adc_buff_1_filled)          // If buffer-1 is still filled -
                    adc_buff_write_enable = FALSE;  // - pause filling until buffer-1 is empty
            }
            else if (adc_buff_write_p == &adc_buffer[ADC_BUFFER_SIZE])  // We are at the end of buffer-1:
            {
                adc_buff_1_filled = TRUE;       // Tell main-loop that buffer-1 is ready
                if (adc_buff_2_filled)          // If buffer-2 is still filled -
                    adc_buff_write_enable = FALSE;  // - pause filling until buffer-2 is empty
            }
        }
    }
    XSysMon_IntrClear(SysMonPtr, IntrStatusValue);  // Clear interrupts from Interrupt Status Register
}
```

(b) ADC's interrupt callback routine

Figure 3.1.9: ADC's interrupt setup- and callback function source code.

When the pointer have come halfway through the buffer a flag is raised telling the main loop that the first half of the buffer is ready to be sent. Finally, when the pointer has filled the last address another flag is raised signaling that the last half is ready to be sent, and the pointer loops back to the first address in the buffer.

The way the sending functions finds the data to transfer is quite similar to the way the data is stuffed in the buffer. Before sending, a struct containing a pointer pointing to the first address of the payload, and the length of the payload is defined. Therefore, even though the payload-data is stored in an array, only the contents of the array is sent and not the array itself.

3.1.4.3 Control unit validation

Different methods were used to validate the operation of the hardware and data gathering. The test to check if the sampling speed was correct was to count how many times "udp_send" was called and for every tenth call to "udp_send" a print was issued.

Another test would point out if the two buffers where filled much faster than data could be transferred. This was done by making a variable that was incremented by one when sending the first buffer and decreased by one when sending the second buffer. As a precaution the interrupt routine can only write to the buffers if a "write_enable" variable is true. In the event that transfer data has not completed sending the buffer before the ADC tries to over-write it, write will be disabled, and a print will alarm the user.

Another fail-safe to alarm one if the program cannot keep up with the frequent interrupts uses the interrupt-controller's instance. The instance includes a counter of how many interrupts is not handled.

It was tested how many buffers python was receiving per second to see if the FPGA was the culprit for the observed error. The setting of test was that the FPGA continuously sent data while python was started at a random time. 1.05 seconds after start the python script stopped gathering data. The number of buffers received was then counted.

While gathering buffers for a set amount of time the data was also put together to form a bigger data set with multiple seconds of data. That big data set was

used to make a WAV file and print the graph to look at the consistency of the waveform.

3.2 Signal processing

This project required several tests in order to better understand the signal processing. Thus the group started by developing simulated tests to see how the different functions perform with a recording from mobile devices of a 440Hz piano note, with 44,1kHz sampling rate. All tough the results may differ between the live test and the simulated test, the pipeline of function use is the same. Therefore, the method part of signal processing focuses on the different functions used in both of the environments.

```
class DC_remover():
    Simeskaa +1
    def mean_value_filter(self, mic_array:list):
        avr_val = sum(mic_array)/len(mic_array)
        # removing the average value from each index
        for i in range(len(mic_array)):
            mic_array[i] = mic_array[i] - avr_val
        return mic_array
```

Figure 3.2.1: Removing DC bias from block of data

3.2.1 Harmonic Product Spectrum

The use of HPS allows the program to calculate what frequency is currently dominating the signal. It also takes into account harmonic copies detected of the sound in the signal, as can be seen in figure 2.3.1 and later on in 4.2.3a. The following function represents the way it was implemented in the final version of the code.

```
23      def HPS2(self, xn, samplerate=44100):
24          L = len(xn)
25          hps_array = []
26          spectrum = np.abs(np.fft.rfft(xn))
27          hps_spectrum = np.copy(spectrum)
28          hps_array.append(spectrum)
29
30          for h in range(2, 5):  # Harmonics from 2 to 4 (adjust as needed)
31              downsampled_spectrum = np.copy(spectrum[::h])
32              hps_array.append(downsampled_spectrum)
33              hps_spectrum[:len(downsampled_spectrum)] *= downsampled_spectrum
34
35          peak_location = np.argmax(hps_spectrum)
36          frequency = peak_location/L*samplerate
37
38          return frequency, hps_array, hps_spectrum
```

Figure 3.2.2: HPS implementation in python

These calculations are performed in the frequency domain, thus the use of $np.argmax(Yk)$ outputs the index of the frequency with the highest peak, which then has to be converted into the actual frequency, as can be seen in line 36 in figure 3.2.2.

3.2.1.1 Signal verification

In order to save processing in the final program, a simple function was design to only perform signal processing when all the different microphones detect the desired frequency. This is done by performing HPS.
The use of HPS is done on all the different microphones independently. The results are then compared to a desired frequency. In the current implementation the frequency is 440Hz. It is to be expected that the frequency detected will vary and not be exactly 440.0Hz. Thus a range is specified with ± 20Hz in the simulated environment, and ± 120Hz in live test. If all of the microphone signals are within the desired range the function outputs *true*, meaning that all the microphones are detecting the desired signal, and the following signal processing is ready to be performed. The implementation of this logic can be seen in the following code 3.2.3. *pro.hps* is the function that calculates the hps of the signal, as explained in 3.2.2.

```
41  def verify_signals(signals, des_Hz: int = 440, width: int = 10):
42      top = des_Hz + width
43      bot = des_Hz - width
44      verified_array = list()
45      freq_array = [0]*len(signals)
46
47      # Populate the detected frequency array
48      for i in range(len(signals)):
49          freq = pro.hps(signals[i])
50          freq_array.insert(0, freq)
51          freq_array.pop(-1)
52
53          # If frequency within range, consider it verified
54          if bot <= freq <= top:
55              verified_array.insert(0, freq)
56
57      # If there are enough verified values, return ready
58      if len(verified_array) >= len(signals):
59          ready = True
60      else:
61          ready = False
62      logging.info(f'worker: Frequencies detected -> {freq_array}')
63      return ready
```

Figure 3.2.3: Signal verification in python

3.2.2 Cross correlation

Cross-correlation is the use of convolution between two signals, with the second signal being conjugated. In this project, this is performed in the frequency domain. It also includes noise whitening and spectral weights in order to perform an accurate correlation. At the very end of the calculation, it is taken into account which of the peaks is the highest, and then the index of which is exported and used to convert it into time delay between the two signals. The function performing this task can be seen in figure 3.2.4.

```
41      # Correlation between two signals, with Fast Fourier Transformation (FFT)
42      def cross_correlation(self, xn_1, xn_2, we):
43          # Convert to frequency domain
44          X1_k = np.fft.fft(xn_1)
45          X2_k = np.fft.fft(xn_2)
46
47          # Cross correlation
48          #R_12 = X1_k*np.convolve(X2_k)                              # Cross correlation (Cc)
49          #R_12 = X1_k*np.conjugate(X2_k)/(abs(X1_k)*abs(X2_k))       # Cc + noise whitening (nw)
50          R_12 = (we*X1_k*np.conjugate(X2_k))/(abs(X1_k)*abs(X2_k))  # Cc + nw + weights
51
52          # Convert to time domain by inverse FFT (IFFT)
53          r_12 = np.fft.ifft(R_12)
54
55          # Calculate time delay between the two signals and distance
56          sample_delay = np.argmax(r_12)
57
58          # In case there is a peak and the wrong side of the correlation
59          if sample_delay > len(xn_1)/2:
60              sample_delay -= len(xn_1)
61
62          # Calculate the time and distance
63          time_delay = sample_delay/self.samplerate
64          distance = time_delay*343
65          return time_delay, distance
```

Figure 3.2.4: Cross-correlation function

This process is performed between microphone 1 and microphone n. This means that microphone 1 is the reference. This can be seen in the code here 3.4.3. A term called Time of arrival delay (TOAD) or Time delay of arrival (TDOA) refers to the time delay between the reading of the different microphones. In other words, time it takes for the other microphones to receive the sound. Unfortunately, it sometimes switches between the two abbreviations throughout the code and this rapport, but they both describe the same values. It is also important to note that an assumption is made that the sound arriving is a straight line.

3.2.2.1 Spectral weighing

In order for the system to be more robust, and not so sensitive to noise, methods introduced in an article [18] were implemented. Noise whitening is performed my dividing the cross-correlation with the absolute sum of the variables multiplied with each other. This can be seen in line 49 in figure 3.2.4. This causes all the different frequencies to have equal amount of impact on the final result, thus weights are calculated and then multiplied with the function as seen in line 50 3.2.4.

A closer look at how the weights are calculated can be seen in figure 3.2.5. This function is quite large, so it has to be broken down.

```
67      def spectral_weighing(self, mic, a, y):
68          nr_mics = len(mic)      #  Amount of microphones
69          nr_samples = len(mic[0]) #  This can be created with only 0, since all mics are same length
70
71          mic_fft = [0]*4
72          mic_fft[0] = np.fft.fft(mic[0])
73          mic_fft[1] = np.fft.fft(mic[1])
74          mic_fft[2] = np.fft.fft(mic[2])
75          mic_fft[3] = np.fft.fft(mic[3])
76
77          # Empty arrays
78          X_k = [0]*nr_samples
79          Xn_k = [0]*nr_samples
80          wk = [0]*nr_samples
81          we_k = [0]*nr_samples
82
83          # Convert the input values to frequency domain
84          #for j in range(nr_mics):
85          #    mic_fft[j] = np.fft.fft(mic_fft[j])
86
87          # Calculate values for every sample between all microphones
88          for i in range(nr_samples):
89              X_k[i] = (mic_fft[0][i] + mic_fft[1][i] + mic_fft[2][i] + mic_fft[3][i]) / nr_mics    # Average
90              Xn_k[i] = max(0.01, np.sqrt( ((mic_fft[0][i] - X_k[i]) +
91                                            (mic_fft[1][i] - X_k[i]) +
92                                            (mic_fft[2][i] - X_k[i]) +
93                                            (mic_fft[3][i] - X_k[i]))**2 / nr_mics ) )  # Std (noise)
94
95              wk[i] = max(0.1, ((X_k[i] - (a * Xn_k[i])) / (X_k[i])) )            # Weight
96
97              # based on function 5 in pdf: https://ieeexplore.ieee.org/document/1248813
98              # we_k (weights with limits)
99              if X_k[i] <= Xn_k[i]:
100                 we_k[i] = wk[i]
101             elif X_k[i] > Xn_k[i]:
102                 we_k[i] = wk[i] * ( (X_k[i])/(Xn_k[i]) )**y
103
104         return we_k, wk
```

Figure 3.2.5: Spectral weights function definition

The input for the function consist of an array of size 4. Where each index holds a buffer size worth of values read by a single microphone. Meaning that $mic[0]$ holds the last 4096 samples from microphone 1, $mic[1]$ holds the last 4096 samples from microphone 2, and so on. All of these signals have to be converted into the frequency domain, since the weights calculated in this function will later be multiplied with the cross-correlation function, which performs it's calculations in the frequency domain. In the for-loop, X_k is the average between all microphones at a given index, Xn_k is the standard deviation between the microphones at the same sample, and if the value is lower than 0.01, its set to that instead. Afterwards wk is calculated with function found in article [18], and can be seen in equation 2.6. Finally we_K can be calculated with the equation 2.7, and can be seen in python figure3.2.5 between lines 98 to 102. we_K is the final weights used as an input in the cross-correlation.

3.2.3 DC Removal

The only environment specific function which is of importance is the DC-bias removal, which can be seen in figure 3.2.1. This is done in order to remove the noise produced by the Analog to Digital Converter (ADC) in FPGA. The function performs a simple subtraction of the average value of the entire buffer from the input, at every single sample, in time domain as stated in theory 2.3.1.

3.3 Angle and distance estimation

It is possible to choose alternative class parameters while initializing the class. It's possible to choose the distance between the microphones, the sound speed, and the maximum distance the distance estimator can calculate. If nothing is supplied, it will automatically utilize a distance of 0.27 meters between the microphones, a sound speed of 343 meters per second, and a maximum distance of 100 meters.

Additionally, it's crucial to remember that the estimators only work with four microphones.

```
ace = angle_cord_estimation(dist_short_mic=0.27563, spd_sound=343, max_distance=100)
```

Figure 3.3.1: Initializing the angle and distance estimator

Additionally, for convenience, all the functions described in this section have been combined into one function. While it is still feasible, it is not required, to use them separately.

```
def timestamp_2_cord(self, timestamps: list):
    # combining the different functions for easier usage
    # ------------------------------------------------
    toad = self.norm_values(timestamps)
    angles, average_angle = self.angle_calc(toad)
    boat_coords_x, boat_coords_y, angle_overrule = self.angle_2_cord_calc(angles, average_angle)
    self.coord_2_distance_calc(boat_coords_x, boat_coords_y)
    return boat_coords_x, boat_coords_y, angle_overrule
```

Figure 3.3.2: Function receiving timestamps and returning coordinates

3.3.1 Angle estimation

The timestamps entered into the routines must begin with the first microphone receiving a signal being zero and the remaining microphones being positive in order for the direction estimate class to perform. Consequently, a method was created to ensure this.

```
def norm_values(self, tdoa):
    fake_tdoa = copy.copy(tdoa)
    norm_tdoa = []
    for i in range(len(fake_tdoa)):
        fake_tdoa[i] *= -1
    low_val_index = np.argmin(tdoa)  # Lowest value
    low_val = tdoa[low_val_index]

    for j in range(len(tdoa)):
        norm_tdoa.append(tdoa[j] + abs(low_val))
    return norm_tdoa
```

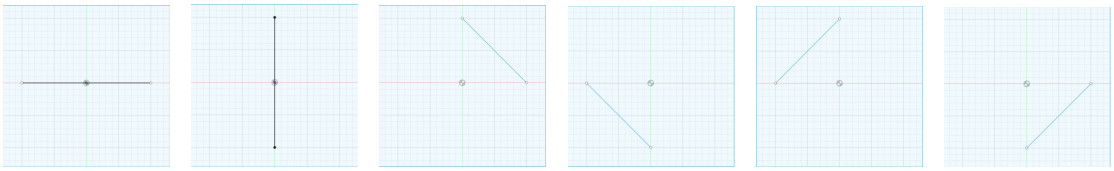
Figure 3.3.3: Function for normalizing the timestamps

There are two distinct ways to determine the angle of the object after normalizing the timestamps. The first technique uses equation 2.8 to compute six distinct angles.

```
# finding angles
# ------------------------------------------------
angle_m3_m2 = np.arccos((fma['m3'] - fma['m2']) * self.spd_sound / self.dist_long_mic) + 0
angle_m1_m4 = np.arccos((fma['m1'] - fma['m4']) * self.spd_sound / self.dist_long_mic) - np.pi / 2
angle_m3_m4 = np.arccos((fma['m3'] - fma['m4']) * self.spd_sound / self.dist_short_mic) - np.pi / 4
angle_m1_m2 = np.arccos((fma['m1'] - fma['m2']) * self.spd_sound / self.dist_short_mic) - np.pi / 4
angle_m3_m1 = np.arccos((fma['m3'] - fma['m1']) * self.spd_sound / self.dist_short_mic) + np.pi / 4
angle_m4_m2 = np.arccos((fma['m4'] - fma['m2']) * self.spd_sound / self.dist_short_mic) + np.pi / 4
```

Figure 3.3.4: Angle calculation with the first method

This code determines the angle between the sound wave and these various lines as shown in figure 3.3.5.



(a) Angle a **(b)** Angle b **(c)** Angle c **(d)** Angle d **(e)** Angle e **(f)** Angle f

Figure 3.3.5: The different angels calculated from the angle estimation

Because the cosine formula is only intended to produce a degree between 0 and 180, first quadrant was used to calculate the angle and then offset to the appropriate quadrant to identify a degree that was higher than 90. Because the maximum and minimum time discrepancies between the various microphones are the same, it is possible to relocate the angle computations to the first quadrant.

To be able to calculate the angle in the first quadrant, the timestamps must be moved to match the timestamps that the first quadrant would receive. Making a duplicate of the timestamp is crucial when working on this to prevent them from changing during the procedure.

```
fake_mics = []
tdoas_temp = copy.copy(tdoa)

for i in range(4):
    ref = np.argmin(tdoas_temp)
    fake_mics.append(tdoas_temp[ref])
    tdoas_temp.remove(tdoas_temp[ref])
```

Figure 3.3.6: Copying time difference values

It's possible that microphones 1 and 2 make initial contact with the sound wave in the first quadrant. As a result, it was required to create some code that took this issue into account and then adds the fictitious time difference to a list so that it matches the values in the first quadrant.

```
# tdoa = time delay of arrivle
m3_m1 = tdoa[2] - tdoa[0]
m3_m2 = tdoa[2] - tdoa[1]
m3_m4 = tdoa[2] - tdoa[3]
m1_m4 = tdoa[0] - tdoa[3]
m1_m2 = tdoa[0] - tdoa[1]
m4_m2 = tdoa[3] - tdoa[1]

sound_diff_1 = ((m3_m1 ⩾ 0) and (m3_m2 ⩾ 0) and (m4_m2 ⩾ 0)) and \
               ((m3_m4 < 0) and (m1_m4 < 0) and (m1_m2 < 0))
sound_diff_2 = ((m3_m1 < 0) and (m3_m2 < 0) and (m3_m4 < 0) and (m1_m4 < 0) and (m1_m2 < 0) and (m4_m2 < 0))
sound_diff_3 = ((m3_m1 < 0) and (m3_m2 < 0) and (m4_m2 < 0)) and \
               ((m3_m4 ⩾ 0) and (m1_m4 ⩾ 0) and (m1_m2 ⩾ 0))
sound_diff_4 = (
           (m3_m1 ⩾ 0) and (m3_m2 ⩾ 0) and (m3_m4 ⩾ 0) and (m1_m4 ⩾ 0) and (m1_m2 ⩾ 0) and (m4_m2 ⩾ 0))

if (sound_diff_1 or sound_diff_2 or sound_diff_3 or sound_diff_4):
    # fake microphone array
    fma = {'m1': fake_mics[0], 'm2': fake_mics[1], 'm3': fake_mics[2], 'm4': fake_mics[3]}
else:
    fma = {'m1': fake_mics[1], 'm2': fake_mics[0], 'm3': fake_mics[3], 'm4': fake_mics[2]}
```

Figure 3.3.7: Rotating the timestamps to the first quadrant

There is an issue when calculating the degree between 0 and 45 degrees for the angle computations for m3-m1 and m4-m2. The issue arises because a negative value is returned when the time differential is found between these microphones. When a cosine is given a negative value, the remaining angle rather than the true angle is returned. It was essential to write some code that corrected the issue in order to go around it.

```
if fma['m1'] > fma['m2']:
    list_angles[4] = np.pi / 2 - list_angles[4]
    list_angles[5] = np.pi / 2 - list_angles[5]
```

Figure 3.3.8: Fixing angle

With the aid of all the timestamps, the second technique of calculation merely computes one angle. It is required to include the location of the microphones and the timestamps, as seen in equation 2.11. Applying pseudo-inverse to the $\overrightarrow{u}$ vector will provide the unit vector. Numpy [23] is used to accomplish this. The system illustrated by the equations is intended for a system with the x-, y-, and z-axes; however, as this system only contains the x- and y-axes, thus the z-axis can be eliminated. In this system the unit vector is calculated to an angle. It's possible to use the method separately and not calculate it to an angle, but to coordinate the methods together this pipeline was enforced.

```
x1 = 0
x2 = self.dist_long_mic / 2
x3 = -self.dist_long_mic / 2
x4 = 0
y1 = self.dist_long_mic / 2
y2 = 0
y3 = 0
y4 = -self.dist_long_mic / 2
T_1 = fma['m1']
T_2 = fma['m2']
T_3 = fma['m3']
T_4 = fma['m4']
c = self.spd_sound

X = np.array([[x2 - x1, y2 - y1],
              [x3 - x1, y3 - y1],
              [x4 - x1, y4 - y1]])
T = np.array([[c * (T_2 - T_1)],
              [c * (T_3 - T_1)],
              [c * (T_4 - T_1)]])
self.U = np.linalg.pinv(X) @ T
angle_center = np.arctan(self.U[1] / self.U[0])

list_angles = [angle_m3_m2, angle_m1_m4, angle_m3_m4, angle_m1_m2, angle_m3_m1, angle_m4_m2, angle_center[0]]
```

Figure 3.3.9: Rotating the timestamps to the desired quadrant

The angles thereafter adjusted to return to the original quadrant. The code in figure 3.3.10 is used to accomplish this.

```
tdoas_temp2 = copy.copy(tdoa)
for i in range(2):
    ref = np.argmax(tdoas_temp2)
    tdoas_temp2.remove(tdoas_temp2[ref])

if (tdoas_temp2 == [tdoa[0], tdoa[1]]):
    pass

elif (tdoas_temp2 == [tdoa[0], tdoa[2]]):
    for i in range(len(list_angles)):
        list_angles[i] = list_angles[i] + np.pi / 2

elif (tdoas_temp2 == [tdoa[2], tdoa[3]]):
    for i in range(len(list_angles)):
        list_angles[i] = list_angles[i] + np.pi

elif (tdoas_temp2 == [tdoa[1], tdoa[3]]):
    for i in range(len(list_angles)):
        list_angles[i] = list_angles[i] + 3 * np.pi / 2

self.grad_list.clear()
for i in range(len(list_angles)):
    self.grad_list.append(list_angles[i] * 180 / np.pi)
```

Figure 3.3.10: Adjusting the correct quadrant

3.3.2 Distance estimation

Angles calculated during angle estimation must be used in the technique to determine the distance from the object. Create lines, then intersect them to determine the object's overall coordinate. Finding the lines' ends is the first thing that must be done. The used equation are: 2.12 .

```
end_cords = {}
for i in range(len(angles)):
    x_name = f'x_{i + 1}'
    y_name = f'y_{i + 1}'
    end_cords[x_name] = self.max_dist * np.cos(angles[i])
    end_cords[y_name] = self.max_dist * np.sin(angles[i])
end_cords['x_7'] = self.U[0][0] * self.max_dist
end_cords['y_7'] = self.U[1][0] * self.max_dist
```

Figure 3.3.11: Making end-points for the lines used in intersection

The line's beginning and ending points must then be determined, as well as their connections. Based on the distance between the microphones, the start positions are determined. The lined are determined in accordance from the left most microphone the angle is made from.

```
start_cords = {'x_1': -self.dist_long_mic / 2, 'y_1': 0, 'x_2': 0, 'y_2': self.dist_long_mic / 2,
               'x_3': -self.dist_long_mic / 2, 'y_3': 0, 'x_4': 0, 'y_4': self.dist_long_mic / 2,
               'x_5': -self.dist_long_mic / 2, 'y_5': 0, 'x_6': 0, 'y_6': -self.dist_long_mic / 2,
               'x_7': -self.U[0][0] * self.max_dist, 'y_7': -self.U[1][0] * self.max_dist}
# making the coords to lines to be used in intersection
# -------------------------------------------------
list_line = {}
for i in range(int(len(start_cords) / 2)):
    name = f'l_{i + 1}'
    p1 = QPointF(start_cords[f'x_{i + 1}'], start_cords[f'y_{i + 1}'])
    p2 = QPointF(end_cords[f'x_{i + 1}'], end_cords[f'y_{i + 1}'])
    list_line[name] = QLineF(p1, p2)
```

Figure 3.3.12: Establishing start-points and making lines with them

The intersecting of the lines can be done in two different ways. The first technique located every line's intersection that wasn't the seventh line. Because the seventh line was created using the second direction technique, which hadn't yet been integrated into the system when the first intersection method was created, the seventh line wasn't intersected using this approach. The seventh line serves as the base for the second intersection method, which intersects all other lines with it. Since they were both equally accurate, the second intersection method was used further in the project.

```
par_12 = list_line['l_1'].intersects(list_line['l_2'])
par_14 = list_line['l_1'].intersects(list_line['l_4'])
par_16 = list_line['l_1'].intersects(list_line['l_6'])
par_52 = list_line['l_5'].intersects(list_line['l_2'])
par_54 = list_line['l_5'].intersects(list_line['l_4'])
par_32 = list_line['l_3'].intersects(list_line['l_2'])
par_36 = list_line['l_3'].intersects(list_line['l_6'])
par_26 = list_line['l_2'].intersects(list_line['l_6'])
par_46 = list_line['l_4'].intersects(list_line['l_6'])

intersection_list_test = [par_12, par_14, par_16, par_52, par_54, par_32, par_36, par_26, par_46]

par_17 = list_line['l_1'].intersects(list_line['l_7'])
par_27 = list_line['l_2'].intersects(list_line['l_7'])
par_37 = list_line['l_3'].intersects(list_line['l_7'])
par_47 = list_line['l_4'].intersects(list_line['l_7'])
par_57 = list_line['l_5'].intersects(list_line['l_7'])
par_67 = list_line['l_6'].intersects(list_line['l_7'])

intersection_list = [par_17, par_27, par_37, par_47, par_57, par_67]
```

Figure 3.3.13: Intersecting the lines

Several indications about the location of the object can be found at the intersection. As a result, the average value of the coordinates is determined, and that coordinate is what the function returns. It's also crucial to remember that some of the coordinates from the intersection are illogical and are eliminated from the average value. The PySide6 [24] library is utilized to accomplish this. When locating junctions, the algorithm additionally reports whether the intersection occurs on the line, is unbound, or don't have an intersection. The coordinate is dropped if it's an unbound intersection. The function will return True for the angle overrule if no points are intersected. It will also provide back a coordinate that is calculated using the average angle and half of the greatest distance between the endpoints. It's vital to remember that if the intersection returns a coordinate may not be completely accurate.

```
bound_inter_list = []
for i in range(len(intersection_list)):
    if intersection_list[i][0] == QLineF.BoundedIntersection:
        bound_inter_list.append(intersection_list[i][1])
x = 0
y = 0
if len(bound_inter_list) > 0:
    for i in range(len(bound_inter_list)):
        x = bound_inter_list[i].x() + x
        y = bound_inter_list[i].y() + y
    x = x / len(bound_inter_list)
    y = y / len(bound_inter_list)
    angle_overrule = False
else:
    angle_overrule = True
    x = self.max_dist * np.cos(average_angle) / 2
    y = self.max_dist * np.sin(average_angle) / 2
```

Figure 3.3.14: Removing unbound intersection coordinates and finding average coordinate

3.3.3 Simulating estimators

It was required to test the system without microphones when constructing the estimators. To achieve this, it was necessary to simulate an object creating sound and determine the various values. It was decided to use three different objects in the test that followed. The first object would be at a distance of 20 meters and 78 degrees. The second object would be 12 meters away and at a 45-degree angle. The final object would be at 35 meters distant and 34 degrees.

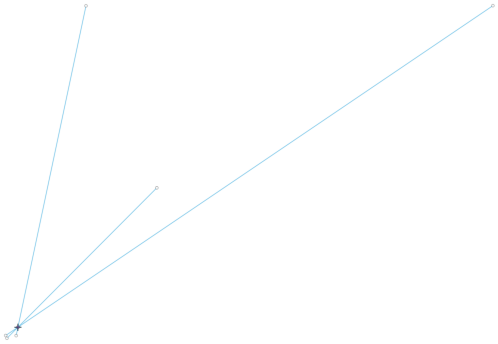


Figure 3.3.15: Overview of the placement of the different objects for simulations

The temporal differentials between the various items have to be calculated after that. The sound wave travel distance between the microphones was discovered in order to determine time differences.

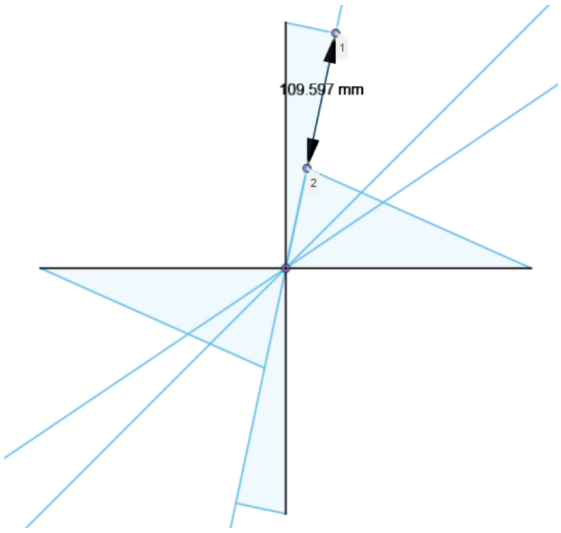


Figure 3.3.16: Sound waves travel distance between microphones

When the distance between the different microphones and the first microphone was found it was possible to calculate the time differentials. This was done with formula 3.2, where ΔT_{1-j} is the time differential between the first microphone and another microphone, ΔD_{1-j} is the distance differential between the first microphone and another microphone and c is the speed of sound.

It was able to determine the time differentials once the distance between the various microphones and the first microphone was established. This was accomplished using the formula 3.2 where ΔT_{1-j} represents the time difference between

the first and second microphones, ΔD_{1-j} represents the distance difference between the first and second microphones, and "c" represents the speed of sound.

$$\Delta T_{1-j} = \frac{\Delta D_{1-j}}{c} \tag{3.2}$$

It was possible to use the same time differentials for all four quadrants because the time differences would be the same in each quadrant, but the microphones receiving them would be different.

3.4 Python

The programming language of choice for this project is python 3.8. While static testing was performed, jupiter notebook was used, but otherwise, pycharm was the main IDE.

In order for the code to be easily manageable, all the different functions were developed in separate files, which then were included in the main program. Github was used to store different version of the code, and enable group members to work separately on the same code.

3.4.1 Main program

Main program has a pipeline of two threads. One is called prod (producer) and the other is called consumer. The two threads communicate with the help of a queue system, and a simple lock, to avoid overwriting data while it's being used. Producer continually reads data sent from the FPGA with the use of UDP protocol. Converts the bit values to actual integers. Normalizes data by dividing the given sample by the max value, and stores the sample in an array. Finally it removes the DC-bias from the signals, and stores the ready signals in an array, which is then placed in the queue. This function can be seen in the following figure 3.4.1.

```
def prod(lock):

    while True:
        data = UDP.get_message(65536)
        a0 = []
        a1 = []
        a2 = []
        a3 = []
        if data is not None:
            for i in range(0, len(data), 2):
                b1 = data[i].to_bytes(1, "little")
                b2 = data[i+1].to_bytes(1, "little")
                if (i % 8) == 0:
                    a0.append(int.from_bytes(b1 + b2, byteorder='little'))
                elif (i % 8) == 2:
                    a1.append(int.from_bytes(b1 + b2, byteorder='little'))
                elif (i % 8) == 4:
                    a2.append(int.from_bytes(b1 + b2, byteorder='little'))
                elif (i % 8) == 6:
                    a3.append(int.from_bytes(b1 + b2, byteorder='little'))
        a0 = DC.mean_value_filter(a0)
        a1 = DC.mean_value_filter(a1)
        a2 = DC.mean_value_filter(a2)
        a3 = DC.mean_value_filter(a3)
        mics = [a0, a1, a2, a3]
        with lock:
            logging.debug("prod  : Got lock")
            q.put(mics)
            logging.info(f'prod  : Put {round(mics[0][0],2)}, {round(mics[1][0],2)}, {round(mics[2][0],2)}, {round(mics[3][0],2)} in queue')
        logging.debug("prod  : lock is free")
        time.sleep(0.01)
```

Figure 3.4.1: Producer function in the main program

Consumer on the other hand reads the data sent from the producer and processes them. This function is too large to have a single figure of, so it will be

broken down in 3 separate parts; Signal verification, signal processing, and distance and angle estimation.

Figure 3.4.2 shows how the main program receives the data from the queue and uses the verification function explained in chapter "Signal Processing 3.2". Start variable is then used to start the signal processing if the desired frequency is detected in all the microphones.

```
97  def worker(lock):
98      empty_vals_array = []
99      """Looping the code"""
100     while True:
101         """Attempt to use the data, if error, give error and kill the while loop"""
102         try:
103             logging.debug("worker: About to ask for lock")
104             """As long as there are values in queue, go through"""
105
106             if not q.empty():
107                 empty_vals_array = []
108                 """Receive the data from queue with lock"""
109                 with lock:
110                     logging.debug("worker: Got lock")
111                     item = copy(q.get())
112                     logging.debug(f'worker: Received {round(item[0][0], 2)} from queue')
113                 logging.debug("worker: lock is free")
114                 logging.debug("worker: starting signal processing")
115
116                 mics = item
117                 """Signal verification"""
118                 desired_Hz = '440'
119                 start = verify_signals(mics, des_Hz=int(desired_Hz), width=20)
120                 logging.info(f'worker: Signal verification determined state {start}')
```

Figure 3.4.2: Signal verification in the main program

Signal processing can be seen in figure 3.4.3. Weights are initially calculated, and then used as an input in the cross-correlation functions. Cross-correlation is calculated between microphone 1 and microphone n as explained in section 3.2. An array of time delays is then normalized and placed in an array.

```
122                 """Signal processing"""
123                 if True: #start:
124                     logging.debug(f'worker: About to calculate weights')
125                     we, wk = pro.spectral_weighing(mics, a=0.3, y=0.4)
126                     logging.debug(f'worker: About to calculate correlation')
127                     t12, d12 = pro.cross_correlation(mics[0], mics[1], we)
128                     t13, d13 = pro.cross_correlation(mics[0], mics[2], we)
129                     t14, d14 = pro.cross_correlation(mics[0], mics[3], we)
130
131                     toad = [0.0, t12, t13, t14]
132                     toad = ace.norm_values(toad)
133                     logging.debug(f'worker: toad estimated to be {toad}')
```

Figure 3.4.3: Signal processing in main program

Last part of the consumer thread is a simple call of a function that converts the time delays, toad array, to an angle and distance. The function call can be seen in figure 3.4.4.

```
136                 # Conversion from time delays to position
137                 logging.debug(f'worker: about to calculate angle and distance')
138                 boat_coords_x, boat_coords_y, dist, average_angle, angle_overrule = ace.timestamp_2_cord(toad)
139                 logging.info(f'worker: Angle {round(average_angle*180/np.pi,2)} degrees')
```

Figure 3.4.4: Use of timestamp 2 coord function in the main code

The remaining code of the consumer thread revolves around breaking the while loop in the case that producer has stopped sending data. After a certain amount of empty values, the while loop will break, resulting in thread finishing.

3.4.2 UDP

To receive the data the FPGA sends, the python script uses a socket API library to catch any message sent on the network interface that was set. The library's receive-function automatically returns the data indexed byte by byte as integers. Therefore an extra step has to be implemented. The data array is passed through a for-loop that converts the data back to bits and splices two and two indexes back together before casting the type to an integer again. The entire conversion is done with byte order "little" meaning the least-significant byte is stored in the smallest address[25].

```python
class UDP:
    def __init__(self, ip_adress: str, port: int, receive_msg: bool):
        # declaring it is a UDP and defining server address
        self.server_address = (ip_adress, port)
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        print(f"Serving connections on port {self.server_address[1]}")
        if receive_msg:
            self.sock.bind(self.server_address)

    def send_message(self, server_msg):
        self.sock.sendto(server_msg.encode(), self.server_address)

    def get_message(self, buf_size):
        data, addr = self.sock.recvfrom(buf_size)
        return data
```

(a) UDP class

```python
while True:
    data = UDP.get_message(65536)
    a0 = []
    a1 = []
    a2 = []
    a3 = []
    if data is not None:
        for i in range(0, len(data), 2):
            b1 = data[i].to_bytes(1, "little")
            b2 = data[i+1].to_bytes(1, "little")
            if (i % 8) == 0:
                a0.append(int.from_bytes(b1 + b2, byteorder='little'))
            elif (i % 8) == 2:
                a1.append(int.from_bytes(b1 + b2, byteorder='little'))
            elif (i % 8) == 4:
                a2.append(int.from_bytes(b1 + b2, byteorder='little'))
            elif (i % 8) == 6:
                a3.append(int.from_bytes(b1 + b2, byteorder='little'))
    a0 = DC.mean_value_filter(a0)
    a1 = DC.mean_value_filter(a1)
    a2 = DC.mean_value_filter(a2)
    a3 = DC.mean_value_filter(a3)
    mics = [a0, a1, a2, a3]
    with lock:
        logging.debug("prod  : Got lock")
        q.put(mics)
        logging.info(f'prod  : Put {round(mics[0][0],2)}, {round(mics[1][0],2)}, {round(mics[2][0],2)}, {round(mics[3][0],2)} in queue')
    logging.debug("prod  : lock is free")
    time.sleep(0.01)
```

(b) Bit splicing

Figure 3.4.5: Receiving data

3.4.3 GUI

It was vital to present the found elements in some way in this thesis. Because the thesis focused on locating boats using sound, it was chosen to create a model of radar to replicate how the process would work if the system were installed on a boat. The QT application or the Python package PySide6 [24] was utilized to present the GUI. The range the GUI can display and the amount of time an object can stay visible on the GUI can both be set when the GUI is initialized. If nothing is set, it will have a showcase of 5 seconds and a maximum distance of 100 meters by default.

```python
window = GUI(max_dist=100, showcase=5)
```

Figure 3.4.6: Initializing the GUI

Additionally, as the GUI is designed to mimic a radar, it uses the same mechanism to display angles rather than using a mathematical method to represent them.

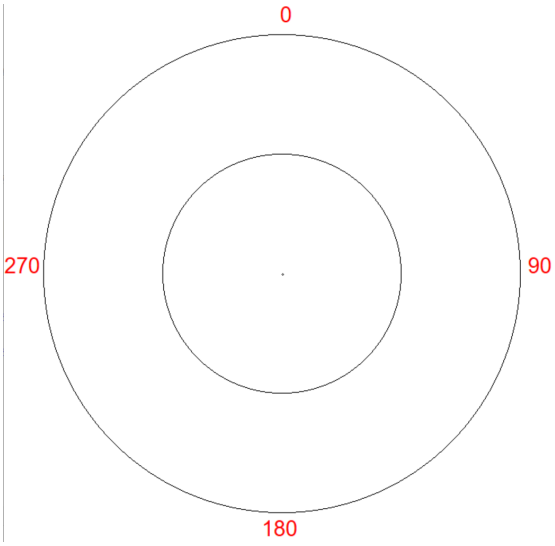


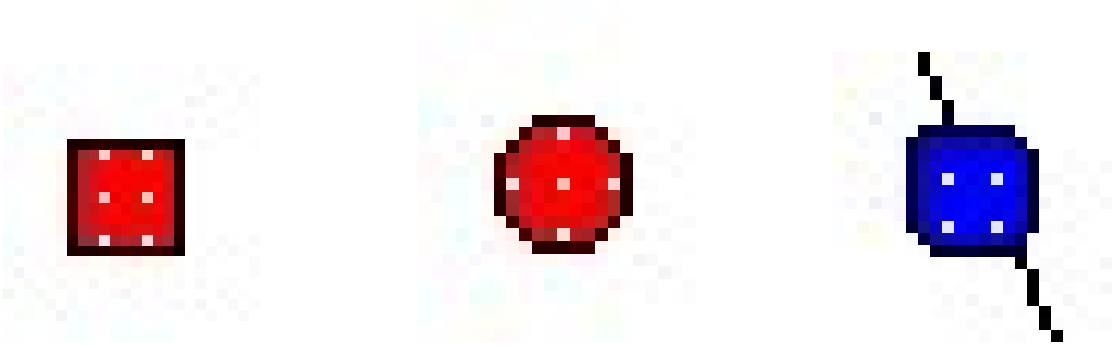
Figure 3.4.7: Overview of the placement of the different objects for simulations

It was chosen to use QPainter from PySide6 [24] to create the GUI. This program made it possible to create a canvas on which to paint the radar and different objects.

```
def radar(self):
    # painting the radar on the canvas
    self.canvas.fill(Qt.GlobalColor.white)
    painter = QPainter(self.canvas)
    painter.setPen(QColor(Qt.red))
    painter.setPen(QColor(Qt.red))
    painter.setFont(QFont('Arial', 20))
    painter.drawText(348, 35, "0")  # x, y
    painter.drawText(660, 350, "90")
    painter.drawText(325, 680, "180")
    painter.drawText(1, 350, "270")
    painter.setPen(QColor(Qt.black))
    painter.drawEllipse(350, 350, 2, 2)
    painter.drawEllipse(200, 200, 300, 300)
    painter.drawEllipse(50, 50, 600, 600)
    painter.end()
    self.label.setPixmap(self.canvas)
    self.update()
```

Figure 3.4.8: Function that paints the radar

The primary function of the GUI is to display the type of object the prototype is finding. Three different sorts of figures are available for the GUI. A squire indicates that distance estimation successfully located an object; a rounded squire indicates that it successfully located an object outside of its maximum distance; and a circle indicates that distance estimation failed to locate the object and is now only able to identify the object's angle. The figures are painted in two distinct colors. The object has a 440 Hz sound wave if the figure is red, and a 260 HZ sound wave if it is blue.



(a) Found coordinate from distance estimation

(b) Couldn't find coordinate from distance estimation

(c) Coordinate out of range for GUI

Figure 3.4.9: The different figures that are able to show on the GUI

You must call the update_GUI function 3.4.11 in order for one of the objects to appear on the GUI. The x, y, HZ, and angle overrule coordinates are required for this function. After gathering the required data, the coordinates must be adjusted to meet the GUI. Although $(0, 0)$ is in the middle for a normal coordinate, it is in the top left corner of the GUI. The coordinate_center function is therefore invoked.

```
def coordinate_center(self, x: float, y: float):
    # centring the coordinates so it fit the GUI
    x_scalar = x / self.max_dist
    y_scalar = y / self.max_dist
    out_of_bound = False

    if x_scalar > 1.:
        x_scalar = 1.
        out_of_bound = True
    elif x_scalar < -1.:
        x_scalar = -1.
        out_of_bound = True
    if y_scalar > 1.:
        y_scalar = 1.
        out_of_bound = True
    elif y_scalar < -1.:
        y_scalar = -1.
        out_of_bound = True

    center_cord_x = (265. * x_scalar) + 350.
    center_cord_y = -(265. * y_scalar) + 350.

    return center_cord_x, center_cord_y, out_of_bound
```

Figure 3.4.10: Centering the coordinate to fit the GUI

The coordinate is not only centered but also checked for out-of-bounds conditions. After centering the coordinate update_GUI 3.4.11 places the coordinates, color, and timestamp in the appropriate figure lists at the time the coordinate was called upon.

```
def update_GUI(self, x:float, y:float, hz:str, angle_overrule:bool):
    self.adding_object = True
    x_adjusted, y_adjusted, out_of_bound = self.coordinate_center(x=x, y=y)
    if not angle_overrule:
        if out_of_bound:
            self.x_rect_circle.append(x_adjusted)
            self.y_rect_circle.append(y_adjusted)
            self.counter_rect_circle += 1
            self.timer_rect_circle.append(time.perf_counter() + self.delay)
            if hz == '260':
                self.red_rect_circle.append(0)
                self.blue_rect_circle.append(255)
            elif hz == '440':
                self.red_rect_circle.append(255)
                self.blue_rect_circle.append(0)
        else:
            self.x_square.append(x_adjusted)
            self.y_square.append(y_adjusted)
            self.counter_square += 1
            self.timer_square.append(time.perf_counter() + self.delay)
            if hz == '260':
                self.red_square.append(0)
                self.blue_square.append(255)
            elif hz == '440':
                self.red_square.append(255)
                self.blue_square.append(0)

    elif angle_overrule:
        # updating GUI, if angle overrule, the angle and distance estimation class couldn't find the distance and
        # only the angle
        self.x_circle.append(x_adjusted)
        self.y_circle.append(y_adjusted)
        self.timer_circle.append(time.perf_counter() + self.delay)
        self.counter_circle += 1
        if hz == '260':
            self.red_circle.append(0)
            self.blue_circle.append(255)
        elif hz == '440':
            self.red_circle.append(255)
            self.blue_circle.append(0)
    self.item_placement_on_GUI()
```

Figure 3.4.11: Function for updating the GUI

The item_placement_on_GUI function is called after the required parameters have been entered. This function displays the most recent coordinate additions on the GUI.

```
def item_placement_on_GUI(self):
    # placing the object on the radar
    if self.counter_square > self.counted_square:
        self.make_square(self.x_square[self.counter_square - 1], self.y_square[self.counter_square - 1], color_index=(self.counter_square - 1))
        self.counted_square += 1

    elif self.counter_circle > self.counted_circle:
        self.make_circle(self.x_circle[self.counter_circle - 1], self.y_circle[self.counter_circle - 1], color_index=(self.counter_circle - 1))
        self.counted_circle += 1

    elif self.counter_rect_circle > self.counted_rect_circle:
        self.make_rect_circle(self.x_rect_circle[self.counter_rect_circle - 1], self.y_rect_circle[self.counter_rect_circle - 1], color_index=(self.counter_rect_circle - 1))
        self.counted_rect_circle += 1
    self.adding_object = False
```

Figure 3.4.12: Adding the newest figure on the GUI

A timer is created when the GUI is initialized and attempts to remove objects every 100 milliseconds. This timer will continue to run while the GUI is active. When the timer expires, the removing_from_GUI method is called.

```
def removing_from_GUI(self):
    # removing the object from the radar
    if not self.adding_object:
        if self.counter_square > 0:
            if self.timer_square[0] < time.perf_counter():
                # deleting the oldest object in the different lists
                self.radar()
                self.x_square.pop(0)
                self.y_square.pop(0)
                self.red_square.pop(0)
                self.blue_square.pop(0)
                self.timer_square.pop(0)
                self.counter_square -= 1
                self.counted_square -= 1

                # adding the remaking objects on the radar again
                for i in range(len(self.x_square)):
                    self.make_square(self.x_square[i], self.y_square[i], color_index=i)

                if self.counter_circle > 0:
                    for i in range(len(self.x_circle)):
                        self.make_circle(self.x_circle[i], self.y_circle[i], color_index=i)

                if self.counter_rect_circle > 0:
                    for i in range(len(self.x_rect_circle)):
                        self.make_rect_circle(self.x_rect_circle[i], self.y_rect_circle[i], color_index=i)

        if self.counter_circle > 0:
            if self.timer_circle[0] < time.perf_counter():
                # deleting the oldest object in the different lists
                self.radar()
                self.x_circle.pop(0)
                self.y_circle.pop(0)
                self.red_circle.pop(0)
                self.blue_circle.pop(0)
                self.timer_circle.pop(0)
                self.counter_circle -= 1
                self.counted_circle -= 1

                # adding the remaking objects on the radar again
                if self.counter_square > 0:
                    for i in range(len(self.x_square)):
                        self.make_square(self.x_square[i], self.y_square[i], color_index=i)

                for i in range(len(self.x_circle)):
                    self.make_circle(self.x_circle[i], self.y_circle[i], color_index=i)

                if self.counter_rect_circle > 0:
                    for i in range(len(self.x_rect_circle)):
                        self.make_rect_circle(self.x_rect_circle[i], self.y_rect_circle[i], color_index=i)

        if self.counter_rect_circle > 0:
            if self.timer_rect_circle[0] < time.perf_counter():
                # deleting the oldest object in the different lists
                self.radar()
                self.x_rect_circle.pop(0)
                self.y_rect_circle.pop(0)
                self.red_rect_circle.pop(0)
                self.blue_rect_circle.pop(0)
                self.timer_rect_circle.pop(0)
                self.counter_rect_circle -= 1
                self.counted_rect_circle -= 1

                # adding the remaking objects on the radar again
                if self.counter_square > 0:
                    for i in range(len(self.x_square)):
                        self.make_square(self.x_square[i], self.y_square[i], color_index=i)

                if self.counter_circle > 0:
                    for i in range(len(self.x_circle)):
                        self.make_circle(self.x_circle[i], self.y_circle[i], color_index=i)

                for i in range(len(self.x_rect_circle)):
                    self.make_rect_circle(self.x_rect_circle[i], self.y_rect_circle[i], color_index=i)
```

Figure 3.4.13: Removing symbols from the GUI

The first symbol in the lists is attempted to be removed each time the removing_from_GUI function is used. The delayed timestamp will be examined using a clock. Nothing will happen if the timestamp is larger than the timer; if the timer is larger, the radar and other symbols on the canvas will be repainted, the first

index in the separate list will be removed, and the canvas will be white-painted. Since QT wasn't designed specifically for creating radars, every time a symbol is removed, there are no utilities in the PySide6 [24] library to erase them. Therefore, if an object on the canvas needs to be removed, the entire piece must be repainted. The memory the GUI uses is unaffected by this solution. Since the canvas creates a collection of pixels and then paints those, memory stacking is not occurring.

CHAPTER

FOUR

RESULTS

The result of the terminal experiment was sub-optimal. The prototype could not give an estimate for the location of the sound source. Hence the project group began analyzing every step of the process, and the findings for each major part will be presented in this chapter, with the result of the experiment presented at the end.

4.1 Hardware

The control unit validation tests came out with the following results:

With a buffer size of 4410 a print happened about every second as expected, although, it was discovered later in the testing-faze that the configured sample-rate was 43706. Because the method to measure the timing was not a very accurate one, it seemed like the print happened every second. However, in theory it would take 1.009 seconds to fill ten buffers, as determined by equation 2.1.

As for the sending mechanism itself, the control-variable came out unchanged after the test meaning the buffer to be sent was alternating as intended. Thereby, write did never get disabled, and the write_enable fail-safe did not activate.

The buffer size was set to 4370 samples per microphone, meaning in theory, ten buffers should be filled per second. Python was set to gather buffers for 1,05 seconds to allow the data transfer routine some time to send the tenth buffer, but still only nine buffers came through.

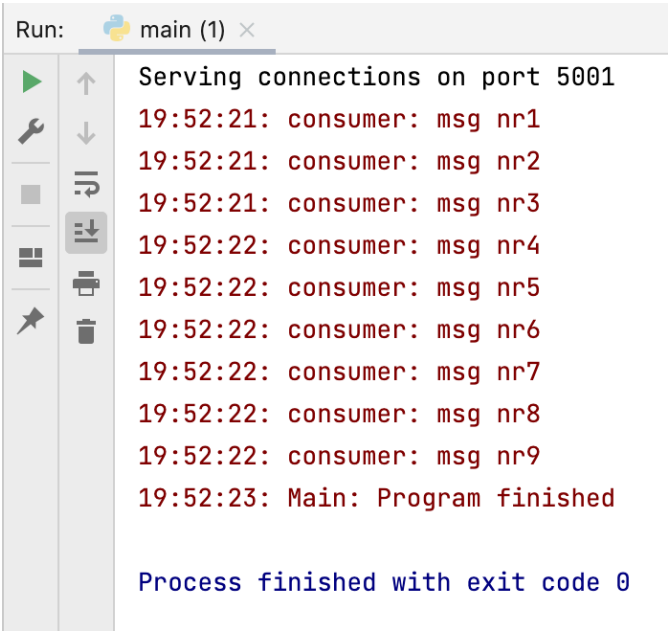


Figure 4.1.1: Captured buffers.

4.1.1 Data analysis

In graph 4.1.2 one buffer has been plotted. Most of the data is noise, but the sound signal can be observed from sample number 3000 and outwards. This indicates that either the microphones should have its sensitivity tuned or the sound should be louder.

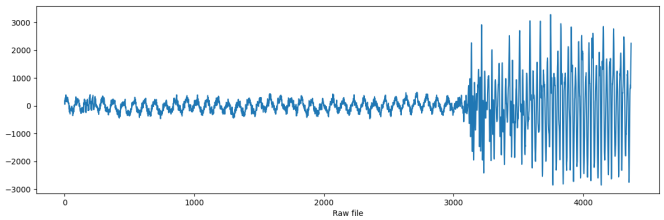


Figure 4.1.2: A single buffer plotted.

When multiple buffers were put together to form a WAV file a distortion was detected. It can be observed in the center of graph 4.1.3. The distortion did not happen at the exact sample where two buffers are put together, but rather about 150 samples later. The full distortion lasted for about 100 samples, and it occurred in every buffer.

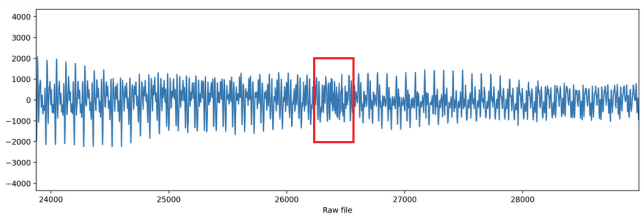
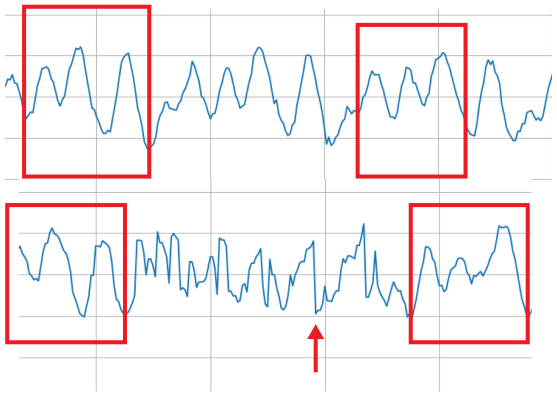


Figure 4.1.3: Multiple buffers assembled into a greater data set.

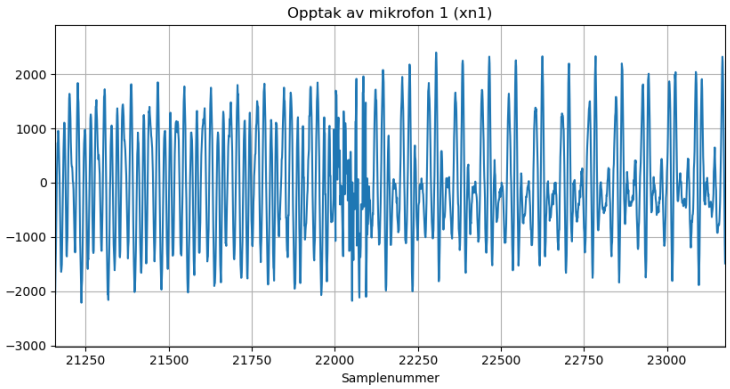
While analyzing the graph formed by the two data sets some patterns emerged. The two graphs in figure 4.1.4a is a zoomed in version of graph 4.1.3. The upper graph show's the distorted part, while the lower graph is an equally zoomed in part right before the distortion. Here it's possible to observe that the wave tops inside the red boxes seems to correspond well between the two graphs. Meaning the upper graph show's what formation to expect to find between the boxes.

Additionally, inside the distorted part it's possible to observe that the graph does some incredibly steep jumps. Where the arrow points to, the graph jumps from a magnitude of 800 till one of -950 in the span of a single sample or 22.8 microseconds.

Furthermore, a similar occurrence can be observed at the same range in figure 4.1.2.



(a) Analyzing the distortion.



(b) Possible time contraction earlier in the graph

Figure 4.1.4: Analyzing figure 4.1.3.

Graph 4.1.4b clearly show's that the signal is different before and after the distortion.

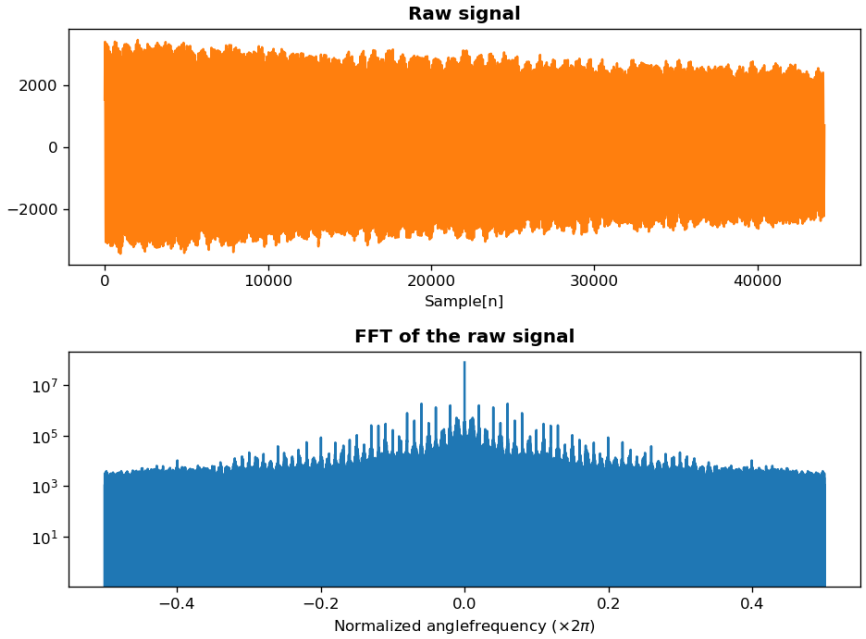
4.2 Signal Processing

The signal processing happens in two separate environments. Simulated static files, and with live data. For ease of readability when discussing the results, each step of the process will be gathered and look at simultaneously. Meaning that the inputs of the system will be placed in one figure.

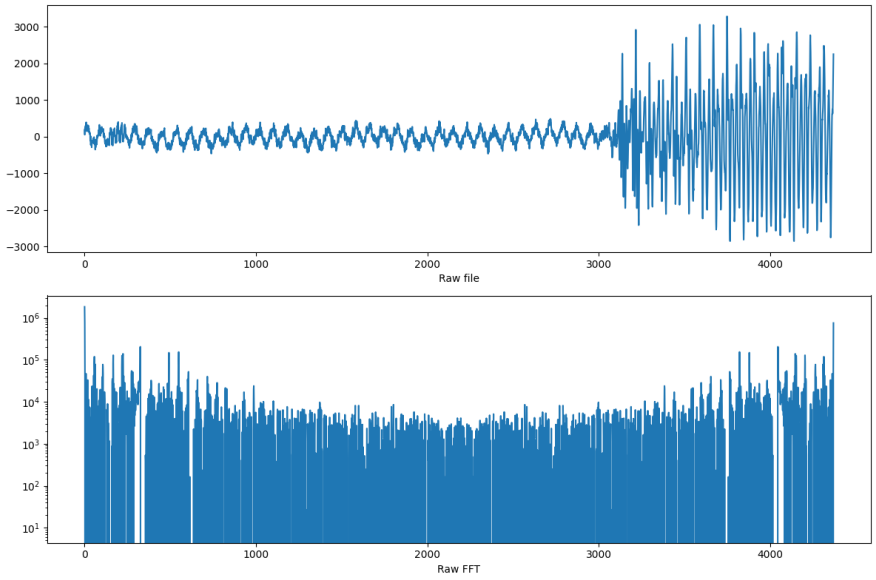
Resulting figures may differ slightly, but the essence of the data is the same. This was due to lack of time, and focus being centered around developing a solution for the problems.

4.2.1 Signal reading

Figure 4.2.1 presents the results of both live and simulated data. The top graph of each figure shows the raw signal in time-domain, while the bottom one is the same signal but presented in frequency-domain, by performing FFT. The main difference between the data is size of the raw signals. In simulated data, the size is 44100 samples, while in live testing its 4410.



(a) Simulated input

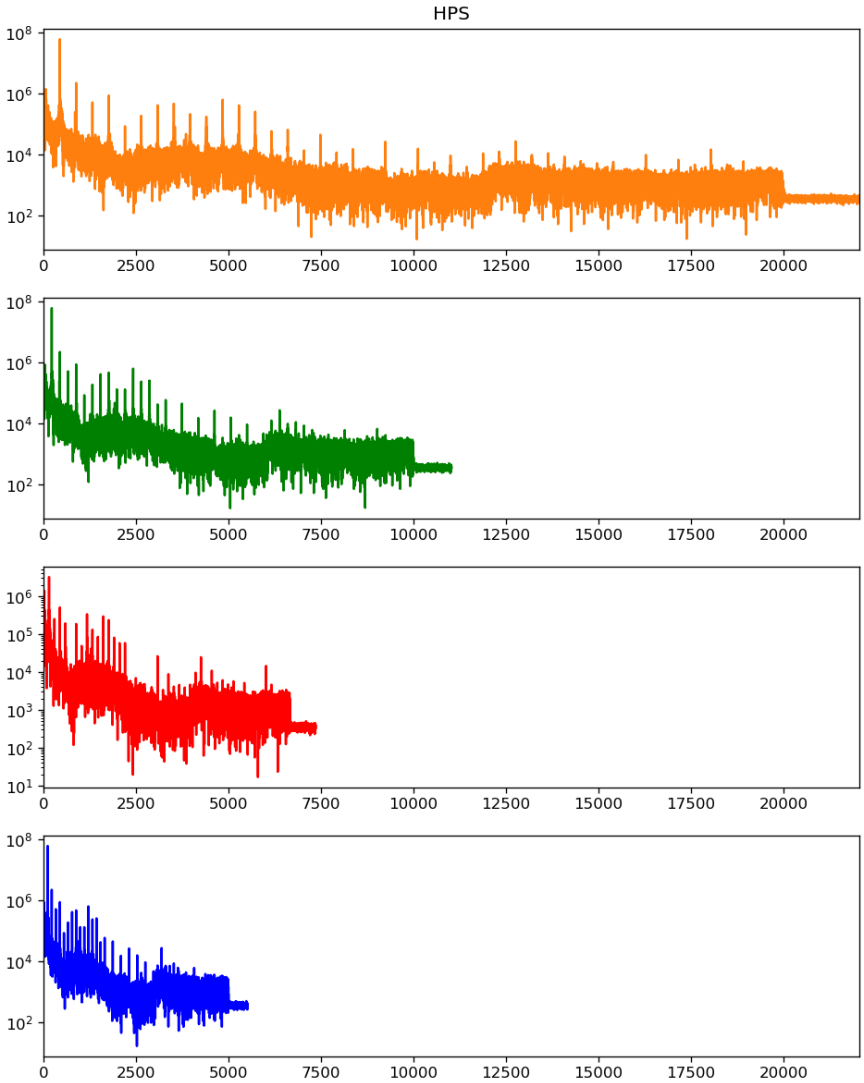


(b) Live input

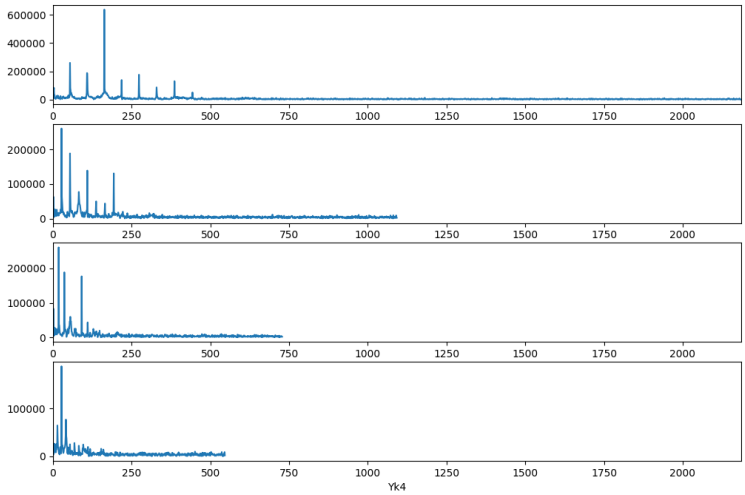
Figure 4.2.1: Reading of one of the microphones of the different systems

4.2.2 HPS

The first step of HPS is to use the frequency domain version of the signal, and downscale it 4 separate times. This can be seen in figure 4.2.2. It is important to note that the simulated data is presented with a logarithmic y-axis, while the live data is not.



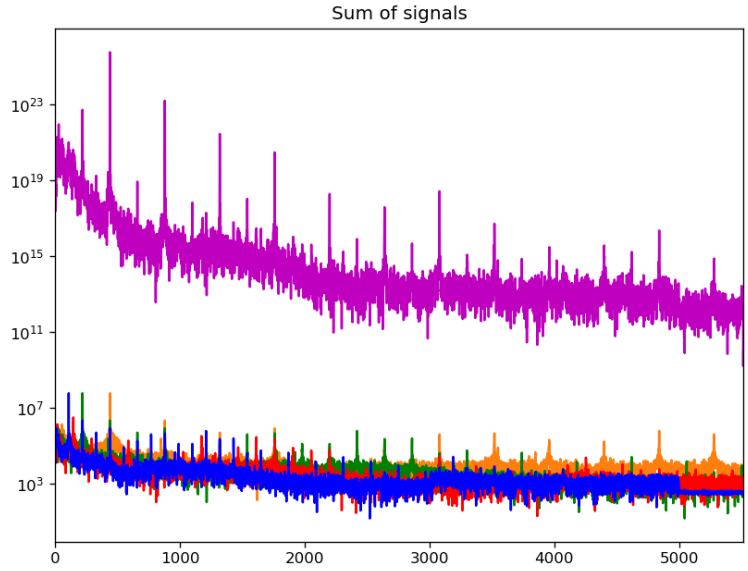
(a) Simulated HPS with an logarithmic y-axis



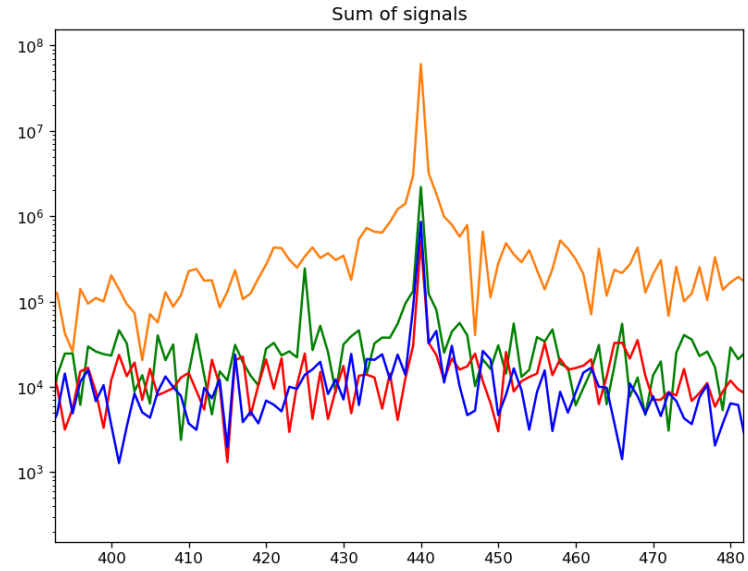
(b) Live HPS

Figure 4.2.2: Scaled harmonic spectrum before summing

The signals are then multiplied with each other. Figure of this process were only printed in simulated test, and can be seen in figure 4.2.3a. The scaled version of the signal can be seen at the bottom of figure 4.2.3a. The summation of these is represented by the magenta colored signal, and will later be used to determine the dominating frequency.



(a) Simulated HPS with overlapping scaled version of the signal and the bottom, and the product presented with magenta color signal

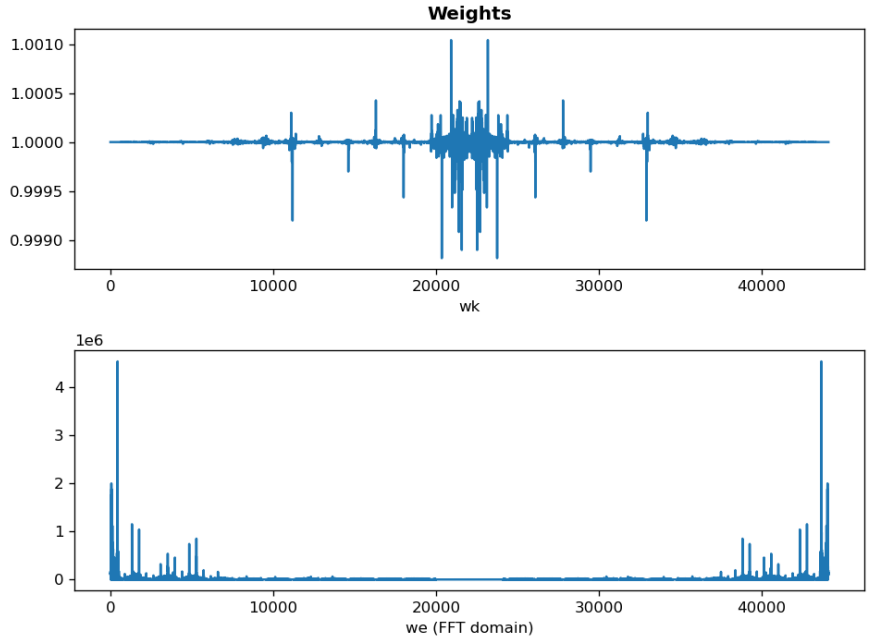


(b) Simulated HPS sum zoomed in at the dominating frequency before sum

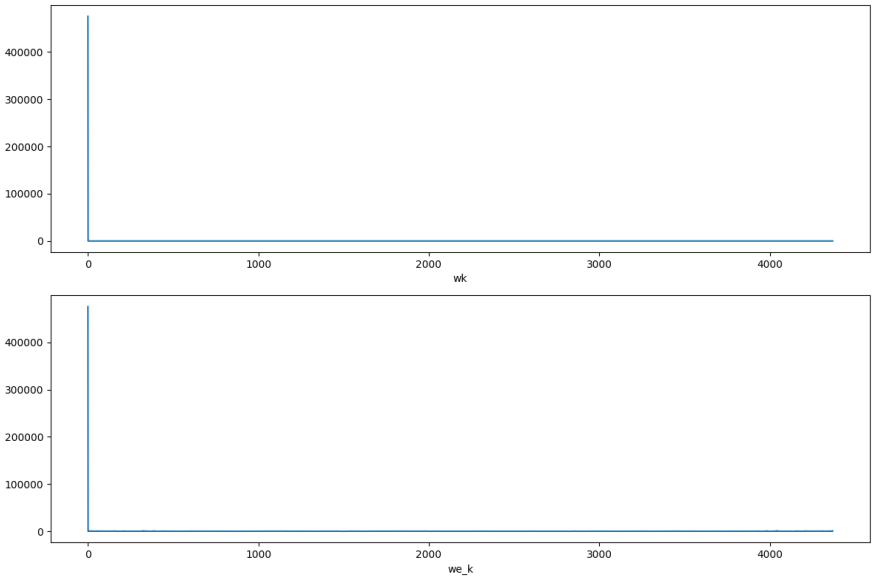
Figure 4.2.3: Scaled harmonic spectrum before summing

4.2.3 Weights

Weights are calculated between all the microphones in a given environment. Meaning that the processes of calculating the weights is only performed once in the system. The result of which can be seen in figure 4.2.4.



(a) Weights for all the different microphones in an simulated environment

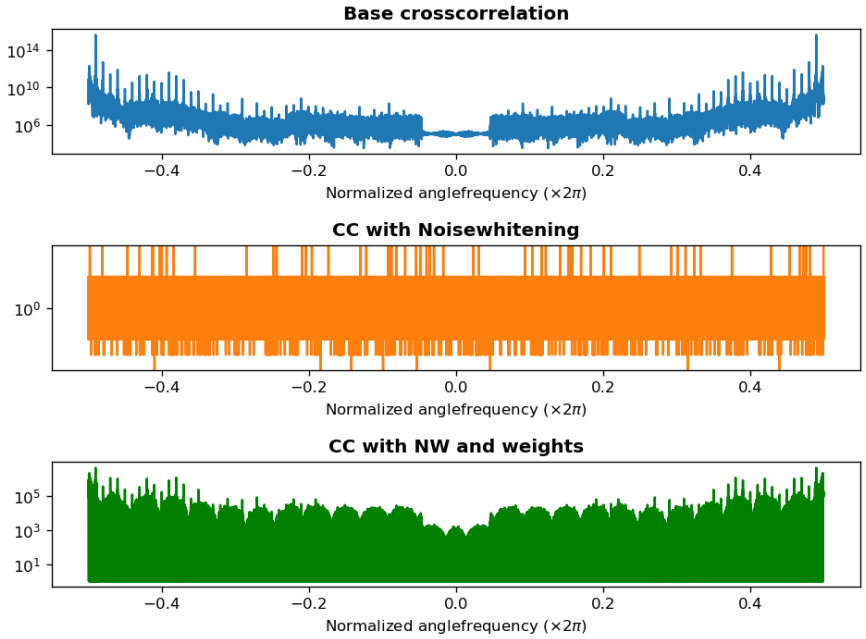


(b) Weights for all the different microphones in an live environment

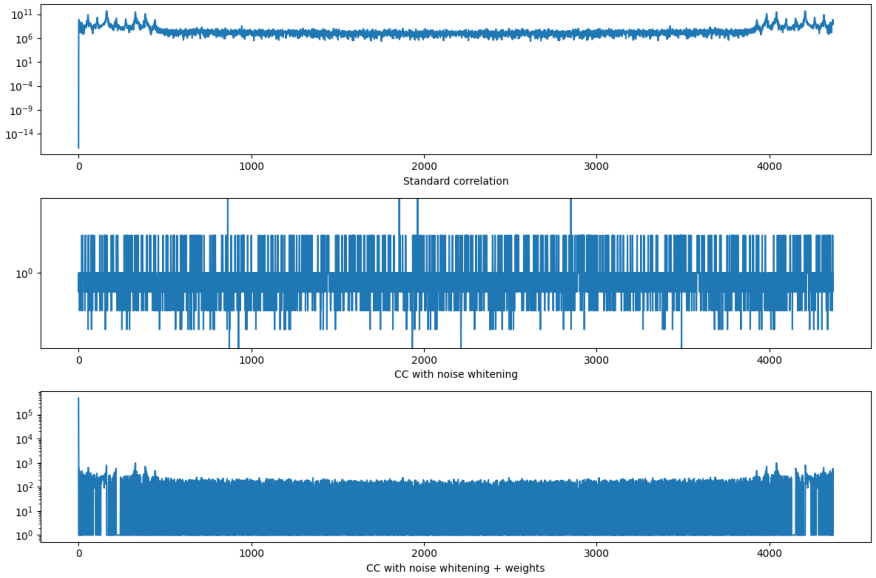
Figure 4.2.4: Spectral weights calculated between all microphones in an given environment

4.2.4 Cross-correlation

There were several different version of cross correlation tested in order to improve the result, thus all of the different methods will be presented in the following figure. Cross-correlation which is used in the final result can be seen in the bottom graph of any given environment figures. Meaning that in figure 4.2.5a, the bottom most graph represents the final cross-correlation used in time estimation.



(a) Different methods of cross-correlation in simulated environment

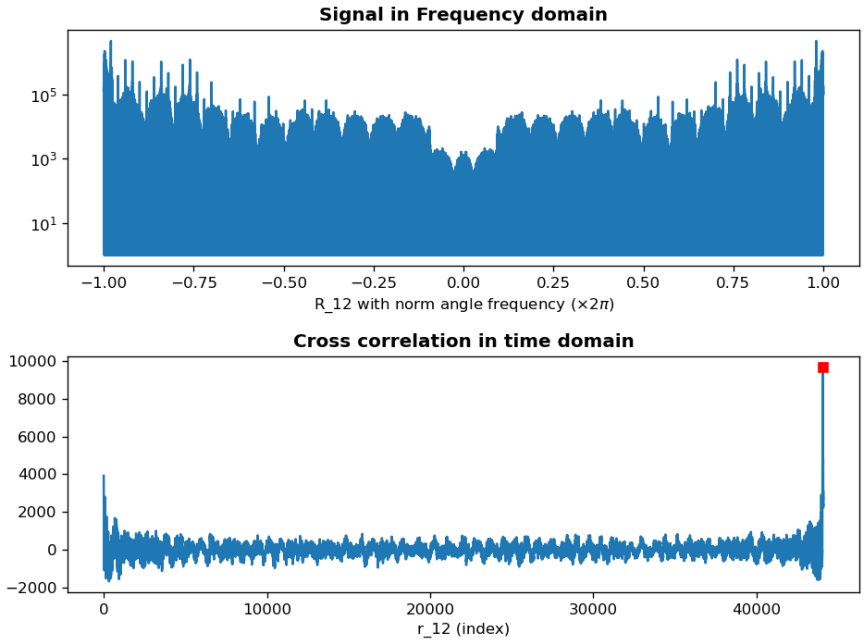


(b) Different methods of cross-correlation in live environment

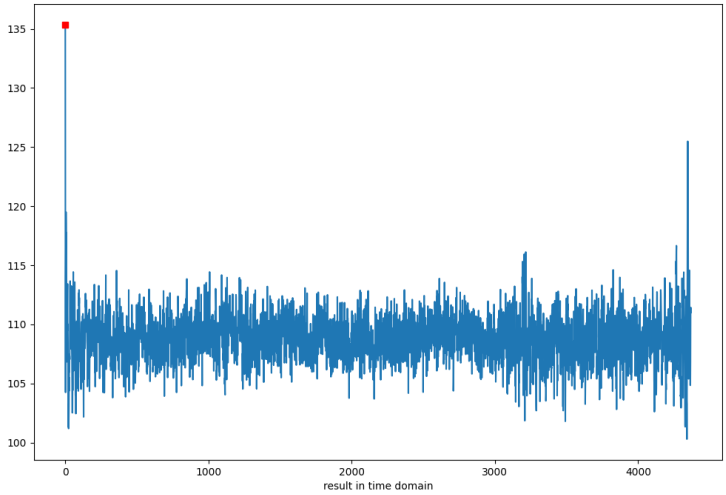
Figure 4.2.5: Visualisation of the different cross-correlation methods in a given environment

The final result of cross correlation can be seen in the following figure 4.2.6. The time-domain graph represents the values that will be used in order to estimate

the time delay between the microphones. The red square highlights the highest peak, meaning, the index where the signals are the most similar.



(a) Cross correlated signal in simulated environment, in time- and frequency domain

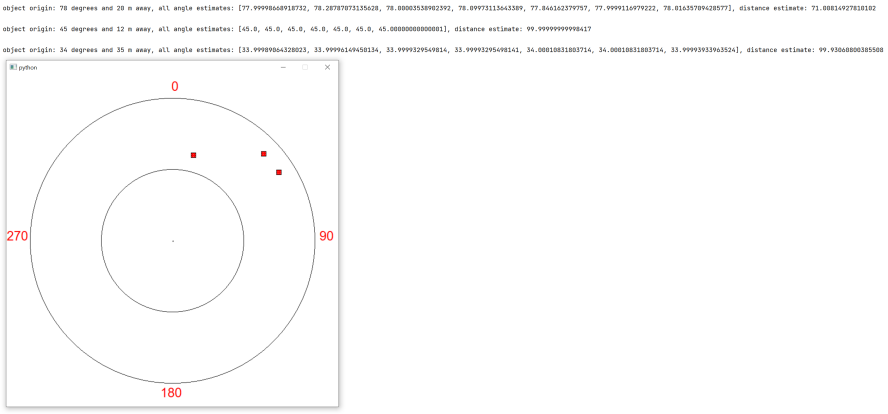


(b) Cross correlated signal in live environment, in time domain

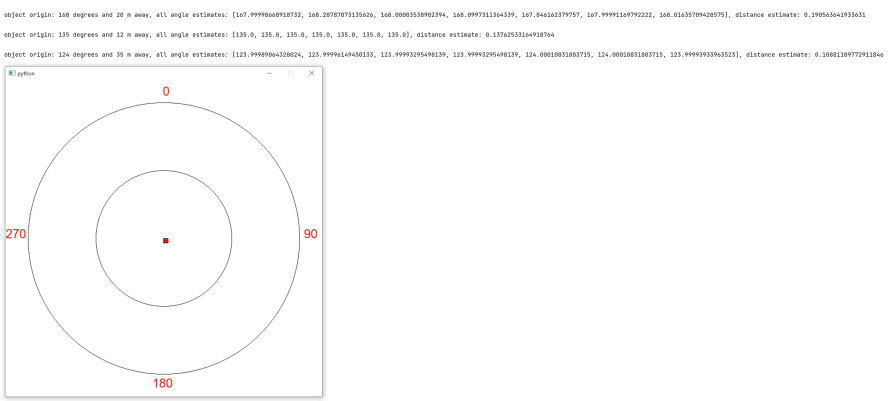
Figure 4.2.6: Final output of the signal processing pipeline. Cross correlated result represented in the frequency domain at the top, and in time domain and the bottom. Live data contains only time domain

4.3 Direction and distance estimation

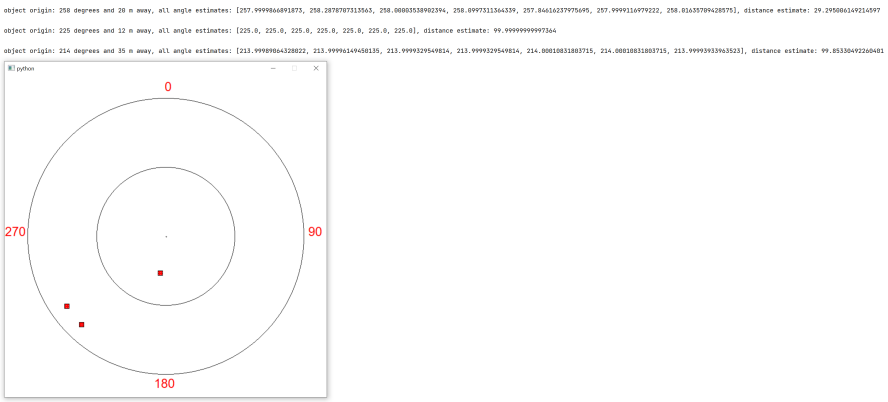
Figure 4.3.1 shows the test results from simulated direction and distance estimation. The figures include the distance and angle of the object, the estimators' estimates, and the location of the object on the GUI. The maximum distance that the GUI and direction estimation can display is 100 meters. The HZ was decided to be 440 HZ because it had no significance because this was a simulated test.



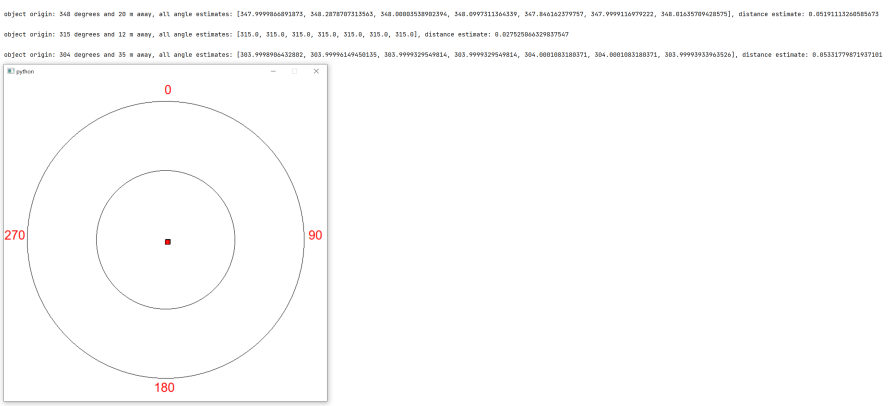
(a) First quadrant



(b) Second quadrant



(c) Third quadrant



(d) Fourth quadrant

Figure 4.3.1: Direction and distance estimation test

As seen in the figures the direction estimate returns a good estimate. Meanwhile, the distance estimate isn't that good.

The estimators didn't receive any positive outcomes from the live testing. The acceptable frequencies breadth was raised from for the estimators' testing. From 440 ± 20 to 440 ± 150 hertz. Additionally, as can be seen in the printout in figure 4.3.2, the incoming time stamps were zero and provided no reliable estimation of direction or distance.

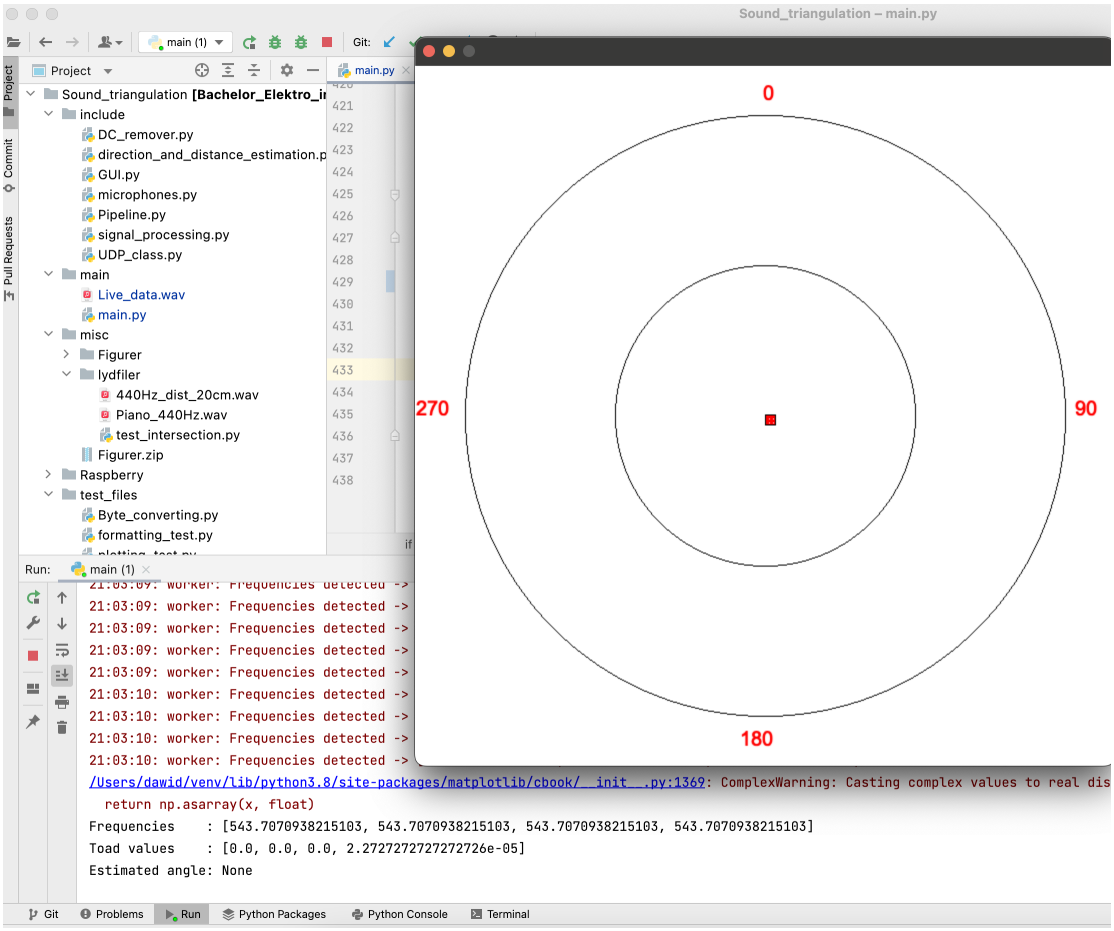


Figure 4.3.2: Picture of platform

CHAPTER

FIVE

DISCUSSION

Main reason for this discrepancy was the lack of time. The group choose to focus on using the available time to better understand the problems of the project, and hopefully find solutions. Causing the simulated data to often be better represented. While some group members attempted to access information from FPGA, the remaining members improved the data available. Which in this case was the simulated data. Main reason this was that it was unsure if the group would manage to read data from FPGA in time, and wanted to have some data to show to, even though a working prototype would not be developed in time for this rapport.

5.1 Hardware

5.1.1 Raspberry pi

Used Raspberry pi as a stepping stone to getting live data, and understanding how data is read from the microphones.

The raspberry was secured with an unknown password. Therefore an OS had to be re-install and setup. Setting up the Raspberry pi meant making one able to access it remotely through an SSH key, and installing the required drivers to access the ADC.

We knew the Raspberry pi would not read data fast enough to make reliable data sets without learning to set up system interrupts. Thus, we chose to not spend time trying to make good data sets with the Raspberry pi. The only value the Raspberry pi contributed with was to see the raw form of data the ADC returned. That information was reassuring when beginning on the FPGA.

5.1.2 Field Programmable Gate Array

It was recommended by the supervisors to use an FPGA. However, none of the members of the project group nor supervisors had any experience with FPGAs, which caused the development of the prototype to be severely bottle-necked.

First of, a little disclaimer; the application won't start sending if it is not given a break-point- and stopped before trying to send the first package on UDP. After the link has been established the debugger can be disconnected and the program

can run freely. This bug was not resolved and we could not find any reason for the application to behave that way, but some digging was done and this is what we know so far:

Before calling "udp_send" for the first time no connection or link has actually been established. Hence when "udp_send" is called the first time it is detected that no link is up, and some netif-routine broadcast's an ARP message asking who has the configured remote IP-address. If the application is not stopped in debug the ARP message is never sent.
The earliest point the code can be stopped is right after the network interfaced is put inside the netif object.

Producing a hardware design was time consuming and not beginner friendly. AMD's tutorials did not give a good understanding of how to design your own circuits and was in some cases actually outdated as well. To check if a hardware design was correct a compiling sequence which would often take upwards of half an hour had to be started. If the design had an error, you'd have to fix the mistake and start the sequence again. Additionally, when compiling a design for the first time it would take about an hour to complete or return an error message. The lack of efficiency and tutorials/examples made the experience very frustrating.

When configuring the ADC to sample at a certain rate it does not tell if the timings of the system allows the frequency or not, making you believe that it will sample at the rate that is given. As a matter of fact it will not always sample at the given frequency, the way to see the real sampling rate a hidden menu in Vitis at the application layer can be opened to observe the real configurations of the hardware. The frustrating part is that if the program tells you if the timing is plausible is inconsistent. If a clock modulator cannot make the frequency specified, a prompt will show up to tell the user that it has automatically been changed.

Through later experimenting with the prototype it was discovered that it is possible to open a menu that shows the hardware configuration though Vitis. This menu showed that the set sampling rate was indeed 174,8 KHz and not 176,4 KHz. Which explained why the sampling rate seemed a little bit off when experimenting the first time.

Because of the nature of the test performed the 16th of may, there is a lot of room for error. The FPGA should use 100 ms to fill a buffer, but because the exact time it takes to actually send the buffer is unknown, we cannot know for sure if some samples falls through or not. Hypothetically if the gathering time was set to 1,1 seconds we'd see 10 buffers, but because we would actually have given it time to fill 11 buffers ten buffers was transferred while some samples still fell through.

It would be very strange for the FPGA to have the possibility to have a sampling rate of one mega samples per second if it is impossible for the microprocessor to keep up with it. The entire reason to use an FPGA is to acquire a system with incredible speed and little latency. Therefore I find our results pretty strange. If we had more time we have a few thoughts that possibly could increase the FPGA's performance that will be discussed in section 5.5; "Future work".

5.1.2.1 Data analysis

Taking into consideration that the ADC can return a magnitude of ± 32768, and the magnitude of what is considered noise is about 500, making the eight least significant bits of the ADC heavily affected by noise. The signal, noise ratio (SNR) shown in figure 4.1.2 is quite bad.

The result accounted for figure 4.1.4a could be accomplished by presenting a significantly higher frequency into the system or by the sampling rate suddenly dropping and becoming unstable.

The hypotheses figure 4.1.4b sparked is that the distortion is indeed the sampling rate becoming unstable. By observing the evolution of the peaks and the density of fluctuations, we can see a waveform that abruptly shifts downwards where the distortion is present. It then continues with its slow shrinking and shifting movement like before. Meaning, that the sampling rate slows down for a certain amount of samples before continuing it's configured rate.

The reason for this to happen can be that while the function transfer_data is called, lwIP begins a sequence of interrupts that has higher priority than sampling the ADC. Thereby making the sampling miss it's timing.

Then again, a counter argument is that because 100 samples is affected by this, lwIP supposedly makes 100 interrupts in order to send data, which we don't know if it does.

5.2 Signal processing

The group lacked experience with live signal processing, and thus made assumptions at the start of the project which resulted in use of development time at functions which would not be used in the final product. One of the main features explored was FIR-filters in order to detect the desired frequency in the signal. It proved to be difficult, and imprecise. After some time, a research paper was discovered by one of the supervisors [18], which was used as a basis for development of cross-correlation, weights and direction estimates.

By comparing the live and simulated data allows a better understanding of what may be errors in the current solution.

As of writing this rapport, the time estimates gathered from the cross correlation are not correct. There are several fields which could resolve the issue which will be further discussed in their respective sections, but the group suspects that the main error comes from either DC-bias removal, or the microphone readings from the FPGA. This will unfortunately impact all of the live data readings, and will make it difficult to observe if the respective functions do their task properly or not. An attempt will be made to at least understand how the initial input error affects the respective sections, and hypothesise their effectiveness by comparing live data with the simulated data.

It's important to note that the graphs in this chapter may look significantly different between live and simulated data, but that is not entirely the case. Some of the graphs are given in logarithmic y-axis in the simulated data, while the live data is given in a standard scalar.

5.2.1 Input

Figure 4.2.1 represents the sounds used in the signal processing. Both of the signals are readings of a 440Hz piano note. The frequency graphs represented are also similar, but the values at each end of the graph are centered. Meaning, that index 0 and 4410 in figure 4.2.1b, are flipped such that they are both placed in the center of the graph as seen in 4.2.1a. A significant thing to note is the magnitude of frequency 0 in figure 4.2.1b, and the center index in 4.2.1a. There is a significant spike in both of the readings. This appears not to be an issue in the simulated data, and thus the group was not surprised to see it in the live data. But as it will be looked into further in the following section, DC-bias usually is detected at frequency 0.

Another thing of note is the resolution of the live signal. The FFT of the signal in simulations 4.2.1a is more consistent and not so noisy. While in the live data in 4.2.1b seems significantly more noisy. The initial assumption was that this was caused by the amount of samples used in the FFT, but even with other test where the size of the processed data, was the same between the live- and simulated environment, the same thing could be seen, but with slightly less impact. The sound was recorded, DC-bias was removed and then the gathered data was converted to a wav file, and then listened to. As explained in FPGA section, the distortion in the signal could heard, but in addition the quality of the microphones were significantly poorer than expected.

A noteworthy discovery while testing different SNRs in the simulation, was that the worse the SNR is, the worse was the quality of resulting FFT of the raw signal. This pattern was then recognized while performing test with live data. With the eight least significant bits being affected by the noise, and the magnitude of the recorded sound signal being $\pm$ 3000, "1011 1011 1000" written as bits. Only the four most significant bits remain unaffected by noise, which can cause inaccuracies, in the form of spectral leakage, when performing FFT.

5.2.2 DC-Bias

The process of removing DC-bias was only required in live data test and thus there are no simulated figures to reference when discussing this section. In the live data however, the dc removal seems to perform as is expected, although there is still something that resembles DC-noise in the input signal. Thus until the FPGA section of this project is resolved it is difficult to get a better understanding of the success of DC-bias removal on the live data.

5.2.3 HPS

It was of interest for the group to develop a function which would only perform most of the calculations when the desired frequency was detected. This was in the early stages attempted by developing a FIR-filter with a band pass around the desired frequency. This solution had quite a few issues, and was inaccurate, and thus the group started searching for other solutions. It was recommended to explore using HPS, but that would restrict the signal detection to be frequency specific. Meaning, that in order to restrain the program from running until the

desired frequency was detected, one had to preemptively know the incoming frequency. This could potentially be worked around by using the magnitudes of the harmonic frequencies detected. By comparing the magnitude the the frequencies detected in static noise, one could create a threshold of peaks detected in noise. Thus if a frequency was above the given threshold, that would mean that something other than noise was dominating the spectrum. This conclusion could be used as a conditional variable in the code, and remove the need of knowing the desired frequency. There are other issues that would have to be solved, and thus, it was not developed in the beginning of this project.

HPS seem to perform quite consistently in the simulated data, as seen in figure 4.2.3a. For a better understanding of how the harmonic peaks overlap, one can look at figure 4.2.3b. Here the signal is zoomed in at index 440 which is the index which is expected to be the place where the most harmonic peaks overlap, since that is the frequency that is produced in the input. This was tested with several different sound inputs and it consistently found the desired frequency. This was not to be the case in the live data, since it consistently missed by a set amount. Meaning that instead of detecting the desired 440Hz and expected, it found the frequency to be around 560Hz. It was hypothesised that this was caused by sample rate error of the FPGA. This was unfortunately difficult to test as discussed in the FPGA section. There are also other reason that could have affected the detection, but since the group could not properly test the sample rate of the input from the FPGA, and other issues were present before this step, no further investigation was done to improve the detection.

5.2.4 Weights

Goal of weights is to improve the cross correlation after it is noise whitened. This will be more thoroughly explained in the cross correlation section of discussion.
As seen in the simulated figure 4.2.4a, the weights are created initially without constrains, wk, and with constrains as seen below in we_k. In the simulated data the weights can clearly be seen throughout the entire frequency spectrum with significant impact. That is not the case in the live data. There is a significant spike at index 0. Before moving forward, its important to note that there are other values in the live result, but they are significantly smaller than the spike at index 0. This was discovered while writing this rapport, and thus a better figure could not be created in time.
The calculated weights are later multiplied by the FFT version of the two microphones that are to be correlated. Meaning that the initial values index 0 will have significant impact, and make the resulting time delay unreliable.

5.2.5 Cross correlation

There are 3 different versions of the cross correlation function used the project. All of which can be seen in both live and simulated data in figure 4.2.5. It is noteworthy that the different steps of the cross correlation look quite similar between the live- and simulated data, beside the value at index 0. This seems to be a continuation of the error already described earlier in this section, and further point to the issue that plagues the results since the start of computation. At this point

the group was certain that there is no further point in developing, or improving solutions, until the input of the system is solved. But from what has been tested with different files in the simulated environment, it is with good certainty that it can be concluded that the cross correlation method presented is supposed to work, given better data in the live environment.

The result of the cross correlation in time domain as seen in figure 4.2.6, is therefore also wrong, and produce a value of either 0.0 seconds, or 2,5*e-5 seconds. Which with certainty can be assumed to be failed. This is further proved by using these values in the distance and angle calculations, which result in a value of None.

5.3 Direction and distance estimation

There were many various ways to estimate direction and distance when the estimators were being developed. The first method that is currently in use and the least square method were the two suggested approaches. It was decided to try the fist technique and see if it was successful. The second method was incorporated into the system later on in the project. The first method has a few drawbacks. Some were corrected, but others weren't. For instance, one of the issues that could be resolved was how the angle was calculated in 360 degrees. Cosines determine a triangle's angle. Therefore, 180 degrees is the largest angle that cosine can calculate. Finding a solution to this was essential.

Calculating the angle in the first quadrant and then offsetting it proved to be the appropriate approach. There were two possible outcomes when the timestamps were moved to the first quadrant. Mics 1 or 2 may be placed nearer to the object than the other. To get around this problem, a system that could locate the microphone that was nearest within a 360-degree circle and then shift the timestamps to the proper microphone had to be created. The answer was to determine whether the various temporal differentials produced positive or negative values. Then construct a boolean logic system around this to determine which microphone was nearest as seen in figure 3.3.7.

Following the various workarounds, a precise angle estimate and imprecise distance estimation was produced in a simulated scenario, as evidenced by the findings 4.3. The real-worlds results aren't discussed because the estimators didn't get any valuable data into the class. It is believed however that the first method doesn't perform as well in a real-world setting. Only two of the four time stamps are used in this method to determine the angle. Therefore, there is a chance that the angle estimation is inefficient if the time stamps obtained aren't very exact.

Additionally, if the time stamps are not right, it may result in incorrect angle estimation and a system fault. The timestamp is converted from a time unit to a distance unit when computing the angle. The cosine is then determined by dividing the calculated distance unit by the distance between the microphones. Only inputs between 0 and 1 can be used to calculate the cosine's return angle. The system will output an error message if the calculated value is more than one. As a result, incorrect time stamps have the potential to generate both a faulty estimate

and an error alert.

The fact that the position is not taken into account while computing the angle in the first method is an additional issue. This is not a problem when estimating the angle alone, but it becomes a difficulty when estimating the distance.

This is due to the methodology used to estimate the distance. The estimate is calculated as a series of lines, each with a start point and an endpoint. The endpoints are computed from the angles, but the angle estimation ignores the start points entirely. As evidenced by the results 4.3, the distance estimate frequently gets absurdly large or small. It was tested with different methods to find the intersection as explained in section 3.3.2, but both returned inaccurate values. The lack of time prevented any further investigation into solutions to this issue.

None of the aforementioned issues exist with the second angle estimation approach. It takes into account the initial point and does not employ any sort of sinus formulae while determining the unit vector. Additionally, it finds the unit vector using every timestamp, making it more reliable. Furthermore, it doesn't have any workarounds and is more apparent as a result. The second approach wasn't tested as much as desired because it was incorporated late in the project. Due to the fact that this approach was created from the scientific article the project is based on, there are probably no significant issues with it. Nonetheless, there may be underlying issues with this method and therefore it can be worthwhile to test for more conclusive results.

One disadvantage of this method is that it only generates a single unit vector. The developed direction estimator cannot be applied to a single vector. Therefore, it is vital to investigate alternative ways of distance estimation if the second angle method is applied.

5.4 Python

5.4.1 Main program

5.4.2 Graphic User Interface

The GUI could have been improved in retrospect. First of all another software could have been utilized to make the GUI, because Qt wasn't the ideal application to use for creating a radar. Once an object has appeared on the GUI, there are no useful functions to remove it. Therefore, a cumbersome piece of code called removing_from_GUI 3.4.13 had to be created in order to remove the figures from the radar.

A significant problem in the GUI was also created due to a lack of experience. When creating a GUI, it's crucial to make all of the modifications in the same thread. There are two threads that modify the GUI. The item placement and removal from GUI functions. A timed function in Qt or Qtimer was used to initiate remove_from_GUI. Each time it is called, the QTimer creates a new thread. There are therefore two threads that modify the GUI. The GUI is in danger of shutting down while it is running as a result of these errors. If the

GUI were to be redone, two classes would likely be required to create it. Two classes—one for printing the variables on the GUI and one that simply updates the symbol variables.

It is significant to remember that the error could not manifest right away. Occasionally, it would surface after five minutes, but frequently, the GUI would function flawlessly for hours on end.

Additionally, the coord_center function as shown in figure 3.4.10 has some flaws. It does not follow the outside circle but offset the symbols in a squire. There are no practical drawbacks to this, but the user may find it puzzling. The usage of equation 2.12 to create the offset to a circle might be feasible, but it wasn't given priority due to lack of time.

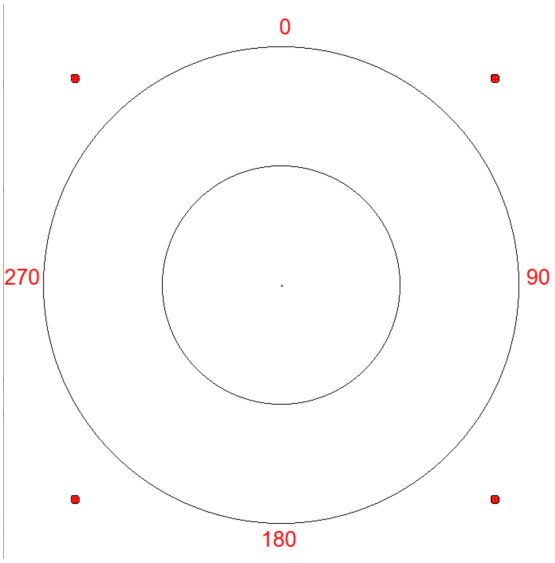


Figure 5.4.1: Square offset to the GUI

5.5 Future work

5.5.1 FPGA

In the case of the ADC's interrupts being de-prioritized because of lwIP. The prioritize could be switched around giving the ADC higher priority than lwIP. Another solution could be adding a FIFO to the hardware design to gather the ADC data through hardware and reading it in bursts. That way the interrupts to gather data will not need to be called as often.

Alternatively only use HDL to make the application. Soft-cores can be found on Github which allows a user to import certain parts of the project instead of having to make everything from the beginning. An example of a soft-core for UDP communication[26] was found on Github. However this approach to the FPGA was discovered after we were done developing the prototype. Therefore the method was not explored further.

Additionally to debug the problem with the network connection read through the lwIP documentation, specifically netif's documentation. Begin by checking if

something for setting up the link is missing. Try to google the problem and see if others have similar experience with the lwIP library. Alternatively try to contact a Xilinx support center. It could be possible to contact "FPGA forum"[27]. One of their committee members gives guidance for Xilinx's FPGAs.

5.5.2 Signal processing

There are several avenues of interest for future work of signal processing. For the project the most significant of which are the following. Gain a better understanding of what causes the raw signal, and the corresponding FFT to be so poor. As mentioned several places in the rapport, an error occurs early on in the processing of the signal, which follows the results throughout the other functions. The current understanding of this issue revolves around the input from the ADC reading is poor, and the sending of the values may interfere with the data. If the issue discovered in the FPGA are solved and the problems within signal processing still persist, the initial place to investigate is DC-bias removal. Thereafter in cascading order one can start solving the different functions, following the pipeline of the project. Meaning, check if the now correct DC-bias removal solved the issues detected in HPS, then move on wards to weights, and so on. The important thing of note is that the signal detection of the project revolves around a specific frequency detection, meaning that the detection is restricted by frequency. Detection can be performed by observing the magnitude of the frequencies peaks detected by the hps. These can be converted to time stamps, or used as detection of signal. The suggested solutions have problems of their own, and thus have to be further investigated.

5.5.3 Direction and distance estimation

A variety of things may be looked into if the estimators were taken further. First, if at all possible, try various approaches in a real-world situation. If a robustness test cannot be conducted in the real world, it may be possible to replicate one. The time differentials are accurate estimates when simulating the values at the moment. In order to examine how the direction estimators function, it could be interesting to add delays to the time differentials and look at the results. This isn't an ideal solution because it's unclear how much of a delay might be introduced into the system. On the other hand, it's still a technique for determining how reliable various methods are but is not a conclusive answer if the methods are valid for real-world situations.

If the first approach performs better than hypothesised, I would also research ways to incorporate microphone coordination into the angle calculation. The distance estimation should improve if this can be added.

Various approaches to estimating the direction and distance can also be investigated. It is possible to estimate the angle using the least square method. Even if there are other approaches that can be taken, the least square method was one of the methods recommended to look into when developing the system but wasn't chosen to look into at that time. For the distance estimation, two microphone sets with a defined distance between them could be used for estimating the distance.

Both sets can make a direction estimation, and it is then possible to either apply a trigonometric function to calculate a distance estimate or to determine the intersection of the lines that were formed from them.

To determine the distance estimate using the present method, it is currently necessary to either look into another direction method or add microphone coordinate to the first direction method. It is important to include a new technique to calculate the distance if the second approach of estimating the direction is continued.

CHAPTER

SIX

CONCLUSIONS

The status of the final project can be summarized as; close to a finished prototype. The entire pipeline of functions interacts with each other, and the result is being visualized. Unfortunately, somewhere between the reading of ADC data and DC-bias removal, an error occurs which makes it difficult to process the data, and produce a satisfying result. Other significant factors that could impact the data readings are SNR.

Based on the test performed, it can also be concluded that the current implementation of second angle estimator is working, while the first method is hypothesized to not be robust enough. First distance estimator is unfortunately not able to produce a satisfactory result either, even with simulated data.

Future work for this project consist of the following. Performing better tests, where the signal magnitude is greater, and causing the SNR to be less of an issue. Resolving the network problem, and periodic distortion in the data sets gathered by the FPGA. Finding definitive evidence of the sampling rate being what it is configured to be. Gaining a better understanding of DC-bias removal, and reason for why the value at index 0 of weights spikes so significantly in live data. Finally, perform live test with the estimators which prove their effectiveness, and find a better solution of distance estimation, by either calculating the multiple angles with a better method, or entirely estimating distance in a different way.

The development of this project, proved that learning how to use a FPGA in this type of project is not necessary given the complexity of the device, power required in computation of the signals, and require more than 3 months to develop for new beginners without supervisors and prior experience. The focus should be shifted to easier to develop devices, like a powerful raspberry-pi, or arduino.

In addition to the technical difficulties, there was another subject in the last semester which greatly reduced the available development time of the bachelor thesis. The group is still proud of the results obtained in the limited time available, and we feel that with more development time the issues currently seen in the final solution could be resolved, and are worth pursuing.

REFERENCES

[1] *Forskrift om forebygging av sammenstøt på sjøen (Sjøveisreglene)*. URL: https://lovdata.no/dokument/SF/forskrift/1975-12-01-5/KAPITTEL_1-4#KAPITTEL_1-4 (visited on 05/14/2023).

[2] *Raspberry Pi Foundation – About us*. en-GB. URL: https://www.raspberrypi.org/about/ (visited on 05/13/2023).

[3] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[4] *What is an FPGA? Field Programmable Gate Array*. en. URL: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html (visited on 05/13/2023).

[5] Per Thorvaldsen. *Grunnleggende Datateknikk*. Vigmostad & Bjørke AS, 2019.

[6] *Principles of data acquisition and conversion*. URL: https://www.ti.com/lit/an/sbaa051a/sbaa051a.pdf (visited on 05/14/2023).

[7] *Introduction and Quick Start • 7 Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide (UG480) • Reader • AMD Adaptive Computing Documentation Portal*. URL: https://docs.xilinx.com/r/en-US/ug480_7Series_XADC/Introduction-and-Quick-Start (visited on 05/14/2023).

[8] "AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite". en. In: (2003).

[9] *Internet protocol suite*. en. Page Version ID: 1152239158. Apr. 2023. URL: https://en.wikipedia.org/w/index.php?title=Internet_protocol_suite&oldid=1152239158 (visited on 05/13/2023).

[10] *lwIP: Overview*. URL: https://www.nongnu.org/lwip/2_1_x/index.html (visited on 05/13/2023).

[11] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. 2006.

[12] *Design and Implementation of the lwIP TCP/IP Stack*. en. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.109.1795&rep=rep1&type=pdf (visited on 05/13/2023).

[13] *Address Resolution Protocol*. en. Page Version ID: 1143868291. Mar. 2023. URL: https://en.wikipedia.org/w/index.php?title=Address_Resolution_Protocol&oldid=1143868291 (visited on 05/18/2023).

[14] *Chapter 22. Protocol Control Blocks - TCP/IP Illustrated [Book]*. en. ISBN: 9780201633542. URL: https://www.oreilly.com/library/view/tcpip-illustrated/020163354X/ch22.html (visited on 05/13/2023).

[15] Richard G. Lyons. *Understanding Digital Signal Processing; Third Edition*. Pearson Education, Inc, 2011.

[16] Tamara Smyth. "*Music 270a: Signal Analysis*". In: (Dec. 2019). DOI: http://musicweb.ucsd.edu/~trsmyth/analysis/analysis.pdf.

[17] Bruce Carter. *Op Amps for Everyone*. Newnes, 2009.

[18] J.-M. Valin et al. "Robust sound source localization using a microphone array on a mobile robot". In: *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*. Vol. 2. 2003, 1228–1233 vol.2. DOI: 10.1109/IROS.2003.1248813.

[19] *Pythagoras equation*. URL: https://www.matematikk.org/artikkel.html?tid=63132 (visited on 05/20/2023).

[20] *Arty A7 Reference Manual - Digilent Reference*. URL: https://digilent.com/reference/programmable-logic/arty-a7/reference-manual (visited on 05/13/2023).

[21] *Getting Started with Vivado and Vitis for Baremetal Software Projects - Digilent Reference*. URL: https://digilent.com/reference/programmable-logic/guides/getting-started-with-ipi (visited on 05/17/2023).

[22] Mohamed A. Bamakhrama. *TCP and UDP echo servers using lwIP RAW API running on Xilinx Zynq Platform*. original-date: 2014-04-15T17:45:47Z. May 2023. URL: https://github.com/mohamed/zynq_echo_servers (visited on 05/13/2023).

[23] *Numpy*. URL: https://numpy.org/doc/stable/reference/index.html#reference (visited on 05/13/2023).

[24] *Qt for python - PySide6*. URL: https://doc.qt.io/qtforpython-6/api.html (visited on 05/13/2023).

[25] *Understanding Big and Little Endian Byte Order – BetterExplained*. URL: https://betterexplained.com/articles/understanding-big-and-little-endian-byte-order/ (visited on 05/14/2023).

[26] *freecores/udp_ip__core*. original-date: 2014-07-16T21:08:26Z. Jan. 2023. URL: https://github.com/freecores/udp_ip__core (visited on 05/18/2023).

[27] *About – FPGA-forum*. en-GB. URL: https://www.fpga-forum.no/about/ (visited on 05/18/2023).

APPENDICES

A - GITHUB REPOSITORY AND WIKI

- Github: `https://github.com/Simeskaa/Bachelor_Elektro_ingenior_med_auto`
- Wiki: `https://confluence.iir.ntnu.no/pages/viewpage.action?pageId=83067491`

B - ATTACHMENTS

- gant plan
- møtereferat
- Forprosjektrapport
- Forslag til innkjøpsliste
- plakat
- midt veis frammføring
- risiko analyse
- Timeliste
- kode
- slutt presentasjon