Chapter 3

Acoustic Positioning System

The acoustic positioning system is the core of the iSBL-SF algorithm. In this system, an acoustic pulse is sent by a transmitter and recorded by receivers on the iSBL array. The time shift between the signals from each receiver is calculated, and these shifts are used to estimate the transmitter's position relative to the iSBL array using an acoustic propagation model. Chapter **??** describes how the orientation estimate of the platform is determined (and how dead reckoning is introduced), and Chapter **??** details how Kalman filters are used to get the best position estimate by combining multiple data sources. This chapter describes only the acoustic positioning system, the central component of this thesis.

The first section in this chapter is dedicated to previous implementations of acoustic positioning systems, and how the iSBL approach differs from them. Next, the mechanical design of the receiver array is discussed. The electrical design of the acoustic positioning system, including the wiring diagram of the full iSBL array, is then detailed. The ultrasonic receivers receive their own section, where the design and manufacturing of the active band-pass filter PCB is explained. Then, the assembly of the full iSBL array is described, followed by the hardware and software design of the transmitter system. The final three sections are devoted to the software of the acoustic positioning system: how an acoustic pulse is detected and recorded, how the time shift between signals is calculated, and finally, how a position estimate is formed.

3.1 Previous Works

Acoustic positioning systems are not a new invention, and many different designs have been tested and implemented over the years. The rise of autonomous underwater vehicles has accelerated research into this field. In this section, the history of acoustic positioning systems is recounted, and the primary difference among these systems (the baseline length) is described. Lastly, similar implementations to this thesis are detailed.

3.1.1 History of Acoustic Positioning Systems

The first known underwater acoustic positioning system was deployed in 1963, when a short baseline acoustic positioning system was used to guide the bathyscape *Trieste 1* to the wreck of the US nuclear submarine USS *Thresher*. The system barely worked, only guiding the *Trieste* to the wreck once after ten unsuccessful attempts. The technology advanced as commercial applications became evident; oil and gas exploration in the 1970s was a primary driver for advancement. In 1998, a long baseline acoustic positioning system was used by a salvage company to navigate a debris field of a sunken WW2 Japanese submarine at 5240m deep. Current advancements in the field are primarily driven by the oil and gas industry, academic research, and defense industry applications [1].

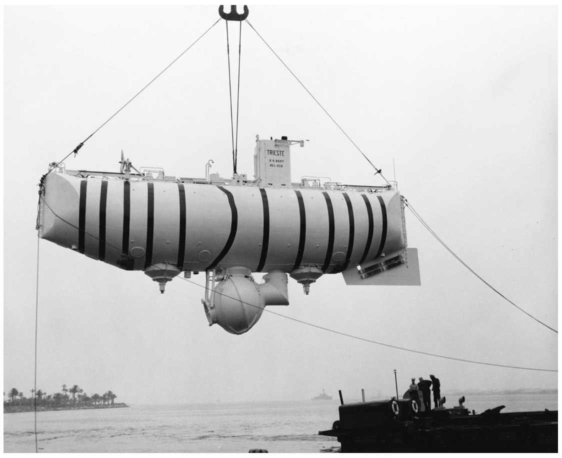


Figure 3.1: Bathyscape *Trieste 1*, which used the first short baseline acoustic positioning system to locate a nuclear submarine wreck [1]

3.1.2 Long, Short, Ultrashort, and Inverted Baselines

All acoustic positioning systems use the same core principle to function: sound waves propagate through a medium in a known and predictable way, and by sending acoustic waves from one or more transmitters to one or more receivers, the relative location between the transmitters and receivers can be determined. The primary way of differentiating underwater acoustic positioning systems is by their baseline, the spacing between receivers (or transmitters, depending on the system).

Long baseline (LBL) systems generally use multiple receivers spaced further than 100m apart in known, fixed locations; a transmitter is attached to the moving object whose position estimate is desired. LBL systems tend to have the best position accuracy when properly calibrated, but require many more resources and much more time to set up [2]. These systems are preferable when working in a single location for long periods of time and when sub-meter accuracy is desired [1]. An additional benefit to LBL systems is that they tend to not require additional measurement equipment such as inertial measurement units or depth sensors, though additional equipment can improve accuracy. See Figure 3.2 for an example of an LBL system.

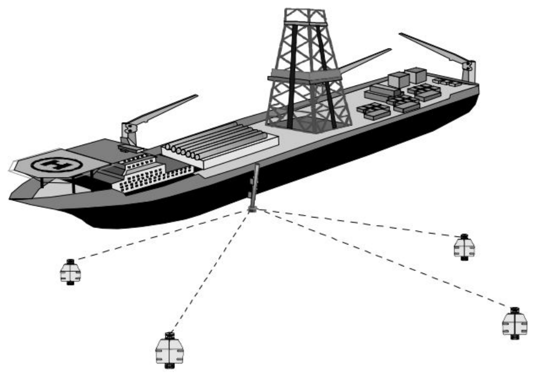


Figure 3.2: Long baseline acoustic positioning system diagram [2]

Ultrashort baseline (USBL) systems generally use a single receiver module with multiple receiver elements spaced less than 10cm apart. These systems usually apply phase-differencing methods to compute the heading between the transmitter and receiver, with the receiver elements spaced less than one wavelength apart [1]. These systems also tend to have low overall complexity and are contained in a single module; however, positional accuracy is less than LBL systems, and additional sensors (orientation sensors, vertical reference units) are required to obtain a position estimate [2]. See Figure 3.3 for an example of a USBL system.

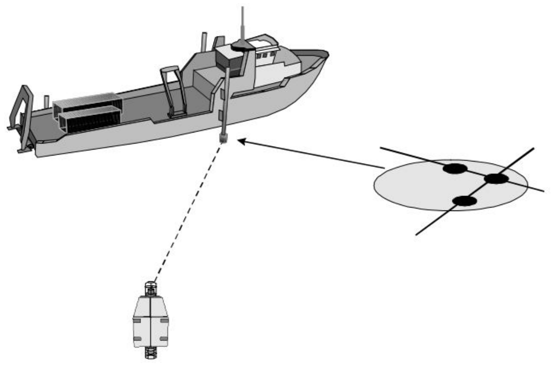


Figure 3.3: Ultrashort baseline acoustic positioning system diagram [2]

Short baseline (SBL) systems are more broad and tend to encompass acoustic positioning systems whose baseline is greater than one wavelength (typically 1m - 100m) and whose receivers (or transmitters) are not fixed to the seafloor [3]. For example, the system of an ROV in the water equipped with a single transmitter and a boat on the surface of the water with receivers on each corner of the hull would be considered an SBL system. SBL systems tend to be a good middle-ground between LBL and USBL: they don't require a fixed receiver array with massive baselines, but can produce position estimates that are close to LBL-level accuracy. These systems do require additional sensors like USBL systems do (orientation and depth), but the receiver array is generally attached to a vessel that already has these systems integrated. See Figure 3.4 for an example of an SBL system.

Inverted acoustic positioning systems flip the location of the transmitter and receiver arrays. An inverted ultrashort baseline (iUSBL) places the transmitter

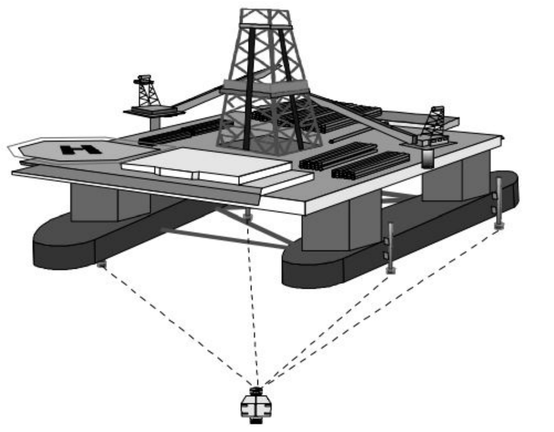


Figure 3.4: Short baseline acoustic positioning system diagram [2]

on the surface of the water (generally on a research vessel or buoy with a known location) and the receiver module on an underwater vehicle - see Figure 3.5 for an example. Due to the small size of USBL system (with the receiver being similar in size to the transmitter), iUSBL systems are quite common, especially in autonomous underwater vehicles [4] [5].

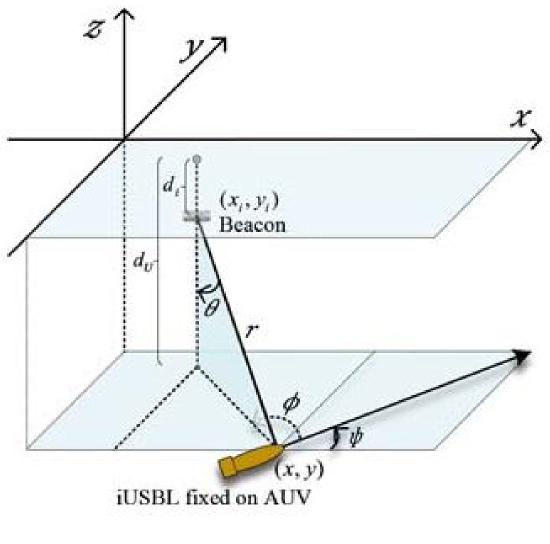


Figure 3.5: Inverted ultrashort baseline system example [6]

Inverted LBL and SBL systems are much more rare; only one paper espouses an iLBL system, whose diagram is shown in Figure 3.6 below. Throughout the research for this thesis, no known papers have been written about iSBL systems - most systems that would use a receiver array on an underwater vehicle tend to space the receivers less than a baseline apart, or otherwise call themselves iUSBL systems.

In this thesis, an iSBL acoustic positioning system is proposed. While iUSBL systems primarily use phase-differencing (but not always) and a baseline of less than one wavelength, this proposed iSBL system will use time-differencing and a

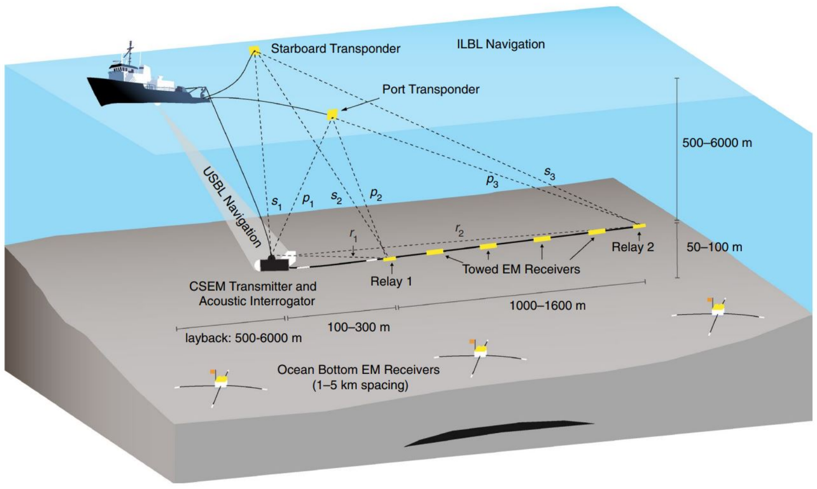


Figure 3.6: Inverted long baseline system example [7]

baseline of greater than one wavelength. The system also uses a single transmitter on the surface of the water (attached to a floating buoy or other autonomous vehicle) and an array of multiple receivers on the underwater vehicle. For a full underwater implementation of this iSBL system, it is recommended to place the receivers as far apart as possible on the body of the underwater vehicle as possible to increase the accuracy of the position estimate.

3.1.3 Similar Implementations

While the classification of this system as an iSBL system is novel, the design of the system draws upon previous research. Since iUSBL and iSBL are quite similar, some iUSBL systems were studied and used as inspiration for this thesis. Luo and Ko of Hoseo University in South Korea designed an iUSBL system with a floating beacon on the surface of the water, and implemented more complex systems (such as simultaneous mapping and localization algorithms) into their design [6]. Researchers from Zhejiang University in China implemented an iUSBL system on an autonomous underwater vehicle that uses FPGAs and beamforming to improve the position estimate [5].

The four most relevant implementations are as follows. Alcocer, Oliveira, and Pascoal from the Institute for Systems and Robotics of Portugal described an acoustic positioning system using GPS Intelligent Buoys (GIBs), which act like a floating LBL system with acoustic and GPS receivers on each buoy [8]. Smith and Kronen of the Florida Atlantic University's Department of Ocean Engineering detailed a low-cost SBL positioning system using a foldable acoustic array [9]. More researchers from Zhejiang University in China created and tested a single-transmitter, variable-baseline acoustic positioning system, the most similar implementation to the one developed in this thesis [10]. Lastly, Morgado, Oliveira, and Silvestre from the Technical University of Lisbon, Portugal tested a USBL system with tightly-coupled inertial measurement devices, which combines the USBL position estimates with an inertial navigation system using an extended Kalman filter [11].

Finally, a GitHub repository by the Underwater Communication and Navigation Laboratory provided many fantastic examples of acoustic positioning implementations. The UCNLNav repository was instrumental in this implementation, giving a great example of implementing Hooke-Jeeves search for time-difference-of-arrival systems [12]. This repository will be revisited later in Section 3.9.2.

3.2 Receiver Array Mechanical Design

The receiver array for this implementation is made of four ultrasonic transducers spaced in an X-pattern, with each transducer having a baseline of 25cm from the center. See Figure 3.7 for an image of the array without any wiring attached. The array is mounted at a 90° angle relative to the Fo-SHIP's end effector, pointing towards the X-axis of the Fo-SHIP.

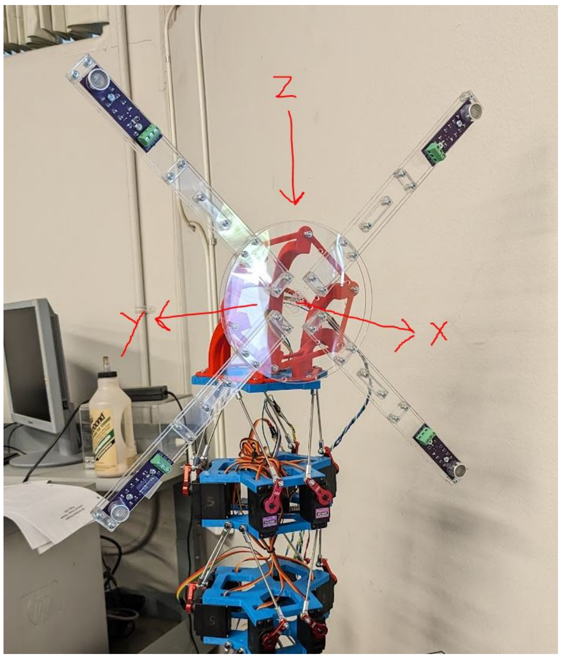


Figure 3.7: iSBL array without wiring or microcontroller, with local coordinate system overlaid

The array is constructed from laser-cut 2.4mm-thick acrylic sheet. The components were designed using SOLIDWORKS and were cut on the Mustang '60 Machine Shop PLS laser cutters. Four long beams are attached to a central circular hub; the microcontroller, soldered breadboard, and IMU have mounting holes on the central hub, while the receivers have mounting holes on the ends of the long beams. The holes are spaced such that the center of the ultrasonic receiver elements are 25cm from the center of the array. A partial CAD model of the array can be seen in Figure 3.8.

Note that the array was constructed from multiple layers of planar acrylic components to reduce the weight of the iSBL array, which affects the overall loading on the Fo-SHIP end effector. The array uses spacer elements to allow for flexibility of the long members, which can help account for any warping of the long, thin elements. Additionally, acrylic sheeting was a low-cost, low-weight, and

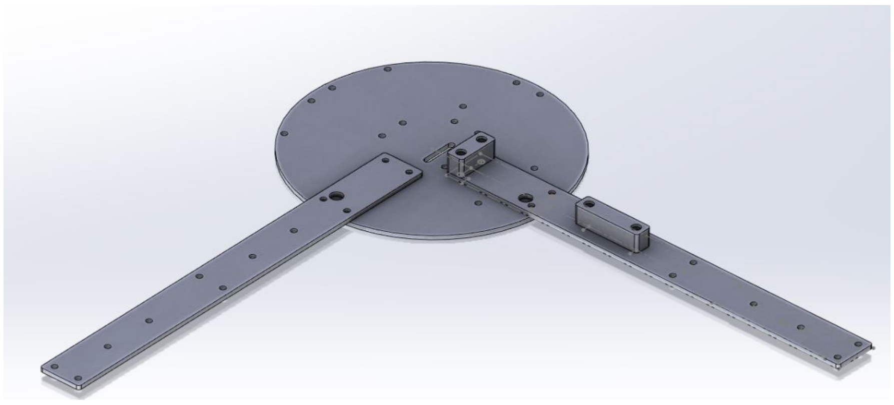


Figure 3.8: Partial CAD model of iSBL array, made using SOLIDWORKS

easy-to-manufacture component when compared to other materials like wood or aluminum. A side view of the array can be seen in Figure 3.9.

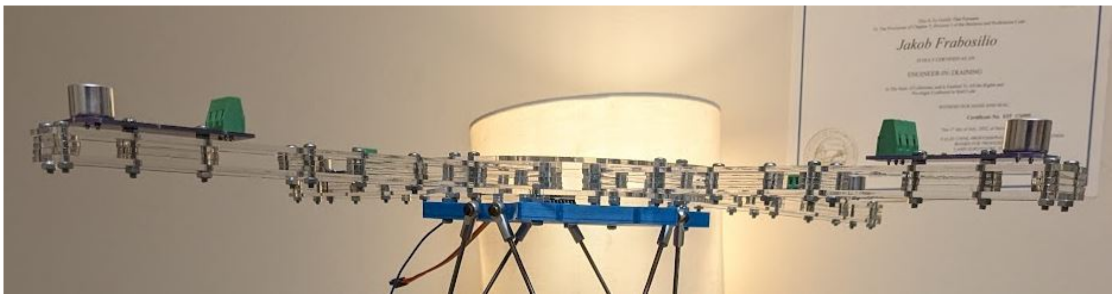


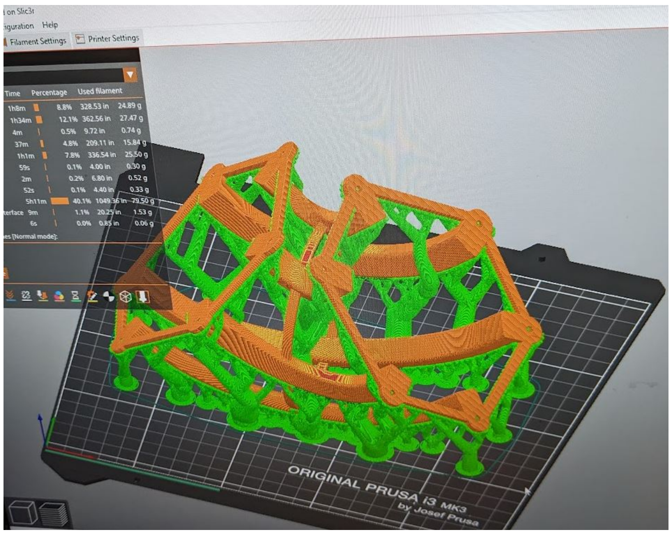
Figure 3.9: Side view of iSBL array

Lastly, a 90° adapter was printed on a Prusa Mk.3 FDM 3D printer in the Mustang '60 Machine Shop. This component is crucial for testing the iSBL-SF algorithm in a horizontal test environment. If the adapter was not added, then the Fo-SHIP would need to be mounted horizontally on a wall (poor loading condition due to gravity), or the transmitter element would need to be suspended in the air above the Fo-SHIP. The adapter was designed in Solidworks and incorporates many lightweighting and stiffening elements. A render of the adapter in 3D printing slicing software can be seen in Figure 3.10.

3.3 Receiver Array Electrical Design

The acoustic positioning system and iSBL array include three main components: the STM32 microcontroller, the receiver elements, and the IMU. The receiver element design is discussed in Section 3.4, and the IMU selection is discussed in Chapter **??**. For the rest of this chapter, it will be referred to as "the IMU," but its full designation is a combined LSM6DSOX 3-axis accelerometer and gyroscope and LIS3MDL 3-axis magnetometer.

The microcontroller used for this project is the STM32H723ZG from STMicroelectronics. This microcontroller was chosen with specific requirements: an analog-to-digital converter (ADC) capable of recording more than 1 million samples per second at 12-bit resolution, random access memory storage greater than



	Time	Percentage	Used filament	
	1h8m	8.8%	328.53 in	24.89 g
	1h34m	12.1%	362.56 in	27.47 g
	4m	0.5%	9.72 in	0.74 g
	37m	4.8%	209.11 in	15.84 g
	1h1m	7.8%	336.54 in	25.50 g
	59s	0.1%	4.00 in	0.30 g
	2m	0.2%	6.80 in	0.52 g
	52s	0.1%	4.40 in	0.33 g
	5h11m	40.1%	1049.36 in	79.50 g
terface	9m	1.1%	20.25 in	1.53 g
	6s	0.0%	0.85 in	0.06 g

es [Normal mode]:

Figure 3.10: iSBL adapter in 3D slicing software (PrusaSlicer)

200kB, digital signal processing libraries, and as high of a CPU clock speed as possible. The H723ZG fulfills all these requirements [13], comes on a development board for easy prototyping and implementation, and is very low-cost at around $30 for one development board [14]. The development board is much larger than required for this system with over 114 GPIO pins and 30 ADC channels, so for future implementations, switching to a custom printed circuit board (PCB) would be recommended. An image of the STM32H723ZG development board can be seen in Figure 3.11.

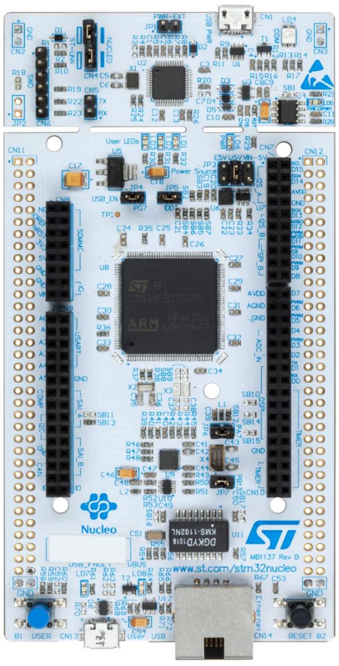


Figure 3.11: STM32H723ZG development board [14]

For this implementation, four ADC inputs, two GPIO pins, one USB connection, one I^{2}C connection, and one serial connection are used. A wiring diagram in Figure 3.12 shows the full schematic of the system. The STM32 communicates with the IMU over I^{2}C, communicates with a laptop via USB, and communicates with the ESP32 controlling the Fo-SHIP via a serial connection. The wiring and communications protocol for I^{2}C and serial are described in Section **??**. Each receiver element is connected to an ADC input, as well as a power and ground connection. Four LEDs are used for debugging and testing; three are internal LEDS on the STM32 development board, while one is connected to an external GPIO pin. A second GPIO pin is used as an interrupt for the IMU; when the IMU has new data ready, the STM32 can detect it via this GPIO pin.

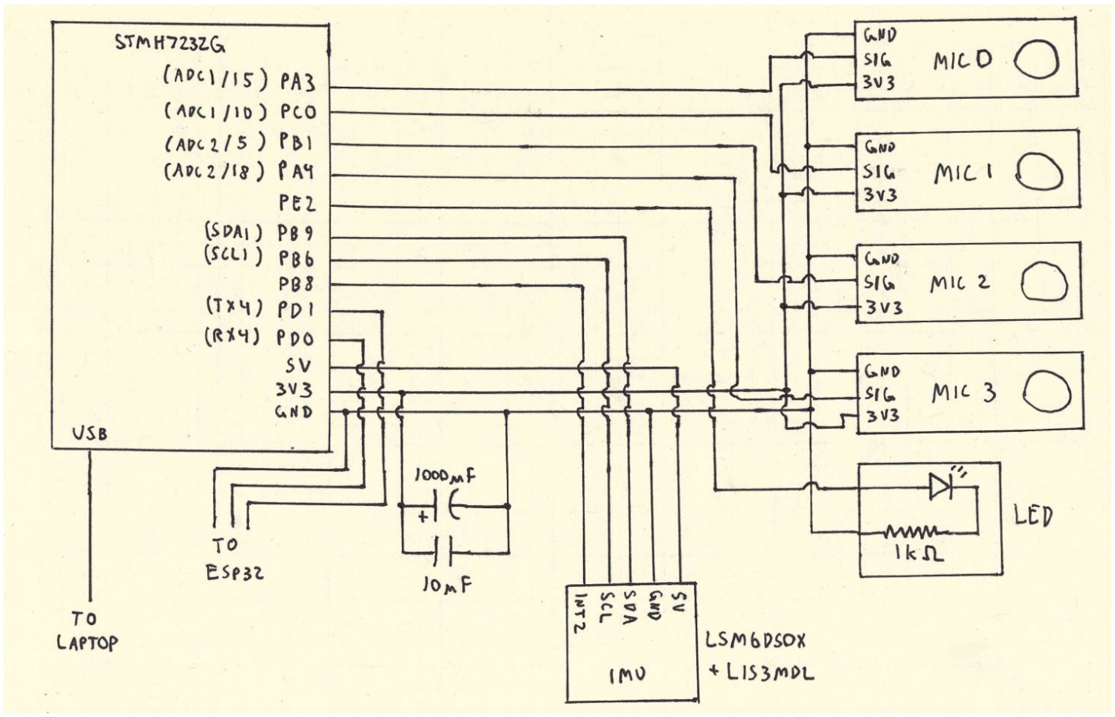


Figure 3.12: iSBL system wiring diagram

Lastly, two capacitors, one large 1000uF electrolytic capacitor and one small 10uF ceramic capacitor, were added to filter out electrical noise from the STM32's 3v3 line.

3.4 Active Band-Pass Filter Design

This implementation is designed to work with 40kHz audio signals. This frequency was chosen for a few reasons:

- The above-water frequency should be similar to the planned underwater system's frequency to allow for simple future implementation
- The audible range should be avoided to minimize disturbance to marine animals (see Section **??** for more details on the underwater system extrapolation)
- Lower frequencies require lower ADC sampling rates to record a single wave
- Lower audio frequencies attentuate much less rapidly in seawater compared to higher frequencies (see Figure 3.13)
- 40kHz is a very common frequency for air-based ultrasonic transducers, and fairly common in water-based ultrasonic transducers

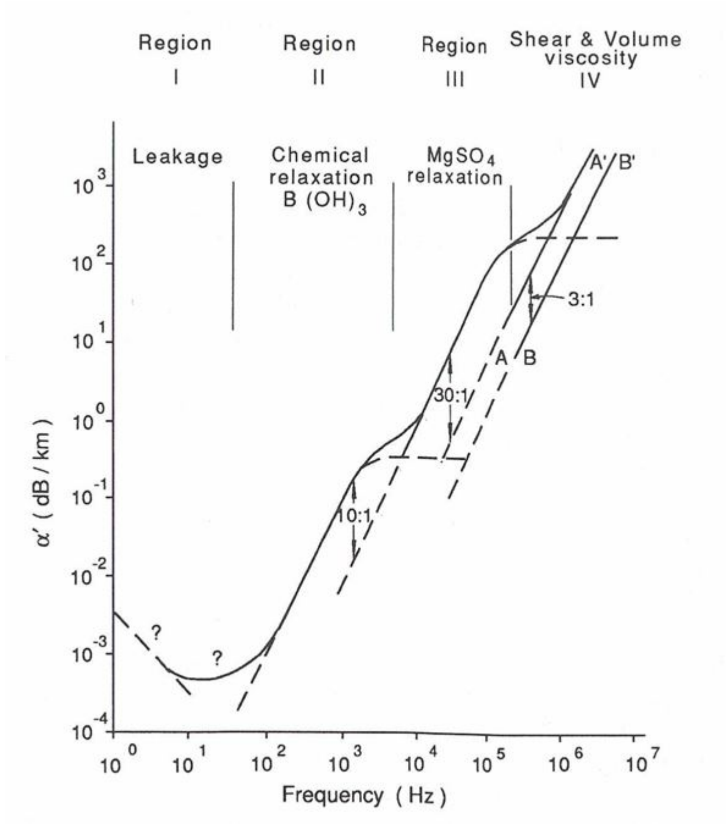


Figure 3.13: Sound attenuation in seawater with dominant attenuation process regions [15]

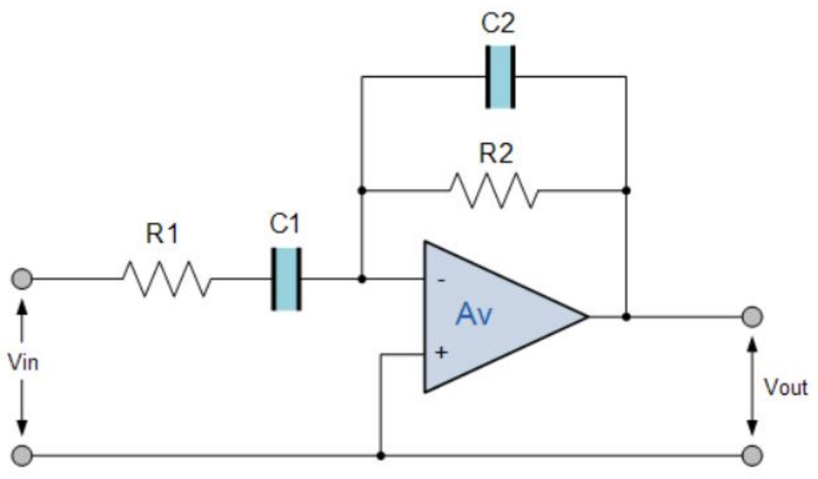
Using ultrasonic transducers without any filtering or amplification would severely limit the range and accuracy of the acoustic positioning system. To overcome both of these issues, an electrical circuit is necessary to:

- Amplify the output voltage of the receiver elements to a sufficient level for the microcontroller's ADCs
- Reject frequencies that are outside of the desired passband
- Be inexpensive and scalable to all microphones
- Reduce the amount of digital pre-processing required

An active band-pass filter is a good solution to the problem and it fulfills the above requirements. Inverting active band-pass filters in particular are designed to have narrow pass-bands. An inverting active band-pass filter based on a circuit from ElectronicsTutorials [16] (see Figure 3.14) was used with the following modifications:

- A decoupling capacitor and voltage divider (C3, R3, R4) that adds a 1.65V DC bias to the microphone signal
- A voltage divider (R5, R6) that sets the non-inverting input of the op-amp to 1.65V

These modifications are necessary for the system to function - see the modified circuit diagram in Figure 3.15. When testing an ultrasonic receiver and transmitter at a distance of 3 meters apart, the receiver registered a peak-to-peak voltage of approximately 10mV when the transmitter was powered by a 12Vpp, 40kHz sine



$$Voltage\ Gain = -\frac{R_2}{R_1},\quad fc_1 = \frac{1}{2\pi R_1 C_1},\quad fc_2 = \frac{1}{2\pi R_2 C_2}$$

Figure 3.14: Inverting active band-pass filter [16]

wave. This voltage oscillates around 0V. The ADC on the STM32 is capable of reading voltages between 0V and 3.3V; to maximize the range of the receiver output, a 1.65V DC bias needs to be added to the receiver signal. Since the op-amp is being powered by an asymmetric 0V to 3.3V supply, the inputs must be biased to the center of the supply voltage range to minimize clipping at the rail voltages. This setup ensures that the whole range of the op-amp is used.

The resistor and capacitor values need to be carefully selected to provide proper filtering and amplification; the equations in Figure 3.14 are the governing equations for this filter. First, a voltage gain of approximately 100 was desired to amplify the 10mVpp signal to 1Vpp; this resulted in a resistor ratio of 100:1. Next, cutoff frequencies as close to 38kHz and 42kHz were desired to reduce the amplification of frequencies outside of this passband. After many iterations of resistor and capacitor values (using common values from the E6 and E12 series to avoid using special components and to reduce cost [17]), the values in Figure 3.15 were chosen. These values result in a gain of -100 and cutoff frequencies of 38.1kHz and 41.3kHz.

The circuit was simulated in LTSpice, an electrical design and simulation software, to validate the design.. A sine sweep test with an AC source placed at the ultrasonic transducer element was run; the results can be seen in Figure 3.16. The simulation shows a peak amplitude response of 34.5dB at 39kHz, an approximate gain of 53. The filter does not have as sharp of a peak as desired, but the sensitivity curve of the ultrasonic receivers used has a similar curve; these two curves add together to produce a frequency response with a large peak at the resonant frequency of the receiver. See Figure 3.17 for the sensitivity curve of the receiver.

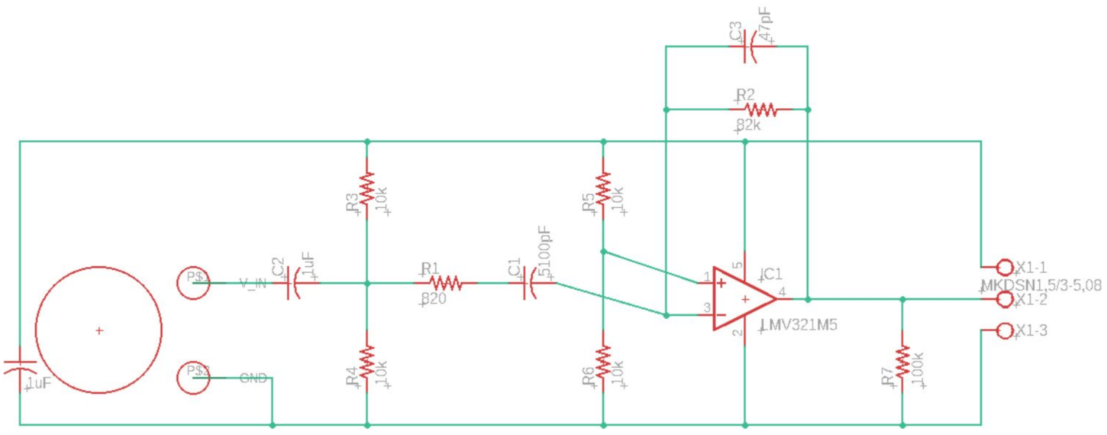


Figure 3.15: Custom active band-pass filter circuit diagram

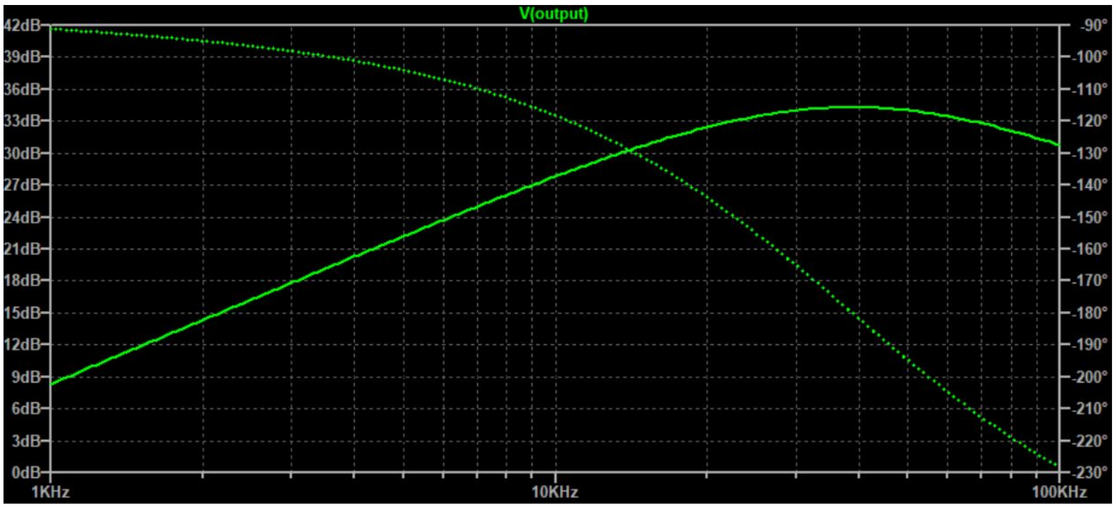


Figure 3.16: Sine-sweep test of active band-pass filter circuit, run in LTSpice. Solid line is amplitude ratio, dashed line is phase shift

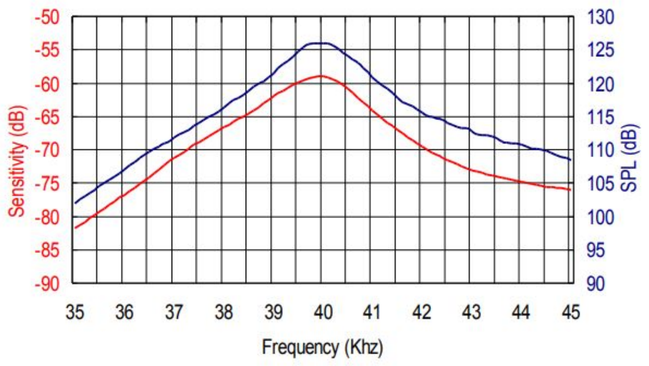


Figure 3.17: Sensitivity of 400SR160 ultrasonic receiver [18]

After selecting electrical components, the PCB files were designed in Eagle (see Figure 3.18) and the PCB was then fabricated by OSH Park (a PCB manufacturer based in Oregon). See Figure 3.19 for the rendering of the PCB from OSH Park. The electrical components for the circuit were purchased from Mouser Electronics.

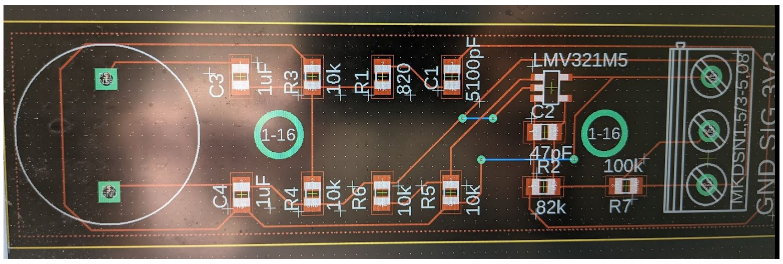


Figure 3.18: Custom active band-pass filter in Eagle

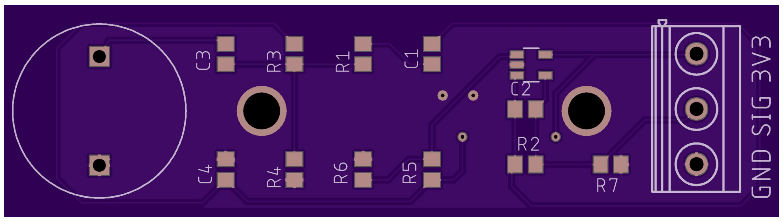


Figure 3.19: Custom active band-pass filter render from OSH Park order

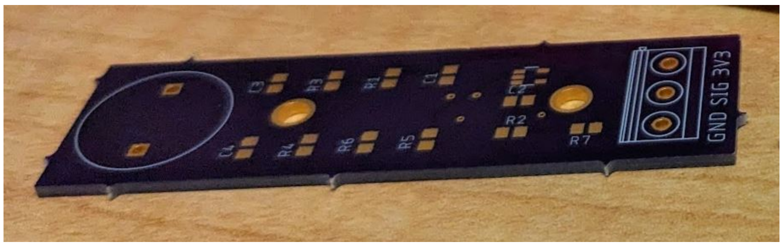


Figure 3.20: Fabricated active band-pass filter PCB, unpopulated

3.5 Receiver Array and PCB Assembly

The assembly process for the iSBL array involved soldering the PCBs and breadboard, assembling the acrylic frame, securing the electrical components to the acrylic frame, and attaching the array to the Fo-SHIP's end effector.

First, the PCBs were assembled. After being delivered from Mouser, the electrical components were soldered to the PCBs; see Figure 3.20 for the PCBs before they were populated with components. All PCBs were hand-soldered using a soldering iron, heat gun, and solder wire. Figure 3.21 shows one PCB in the process of being soldered, and Figure 3.22 shows a completed PCB.

After this, the noise-reducing capacitors were soldered to a solderable breadboard. Header pins for each PCB signal line, as well as corresponding wires, were also soldered to the breadboard. Figure 3.23 shows the breadboard. Note that three wires go from the breadboard to each PCB: ground, signal, and 3.3V power.

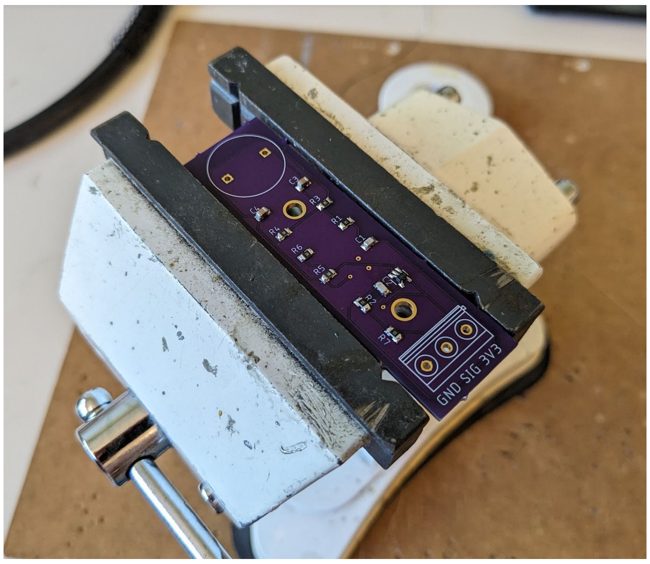


Figure 3.21: Soldering in-progess for PCB

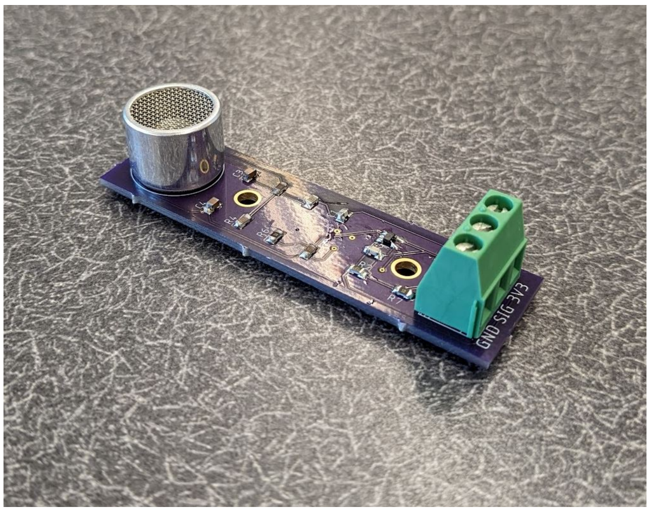


Figure 3.22: Custom active band-pass filter render from OSH Park order

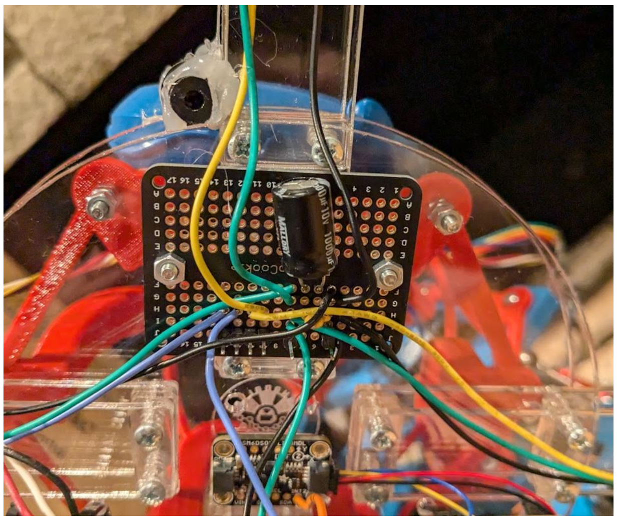


Figure 3.23: Soldered breadboard with capacitors, header pins, and wires to ultrasonic receiver PCBs

Next, the acrylic frame was assembled using the laser-cut acrylic components, M3x16 screws, and M3 nuts. Spacer plates were sandwiched between the long receiver arm plates and the central hub plates. The receiver PCBs and the soldered breadboard were secured to the acrylic using the M3 screws and nuts. Mounting holes for the IMU and the STM32 development board were not accessible during assembly due to poor planning - the STM32 is secured to the acrylic hub using cable ties, and the IMU is secured to the acrylic hub plate using adhesive. Two cable tie segments were cut from a single tie and placed on either side of the base of the IMU, and adhesive was placed in the space between the segments. This ensures that the IMU is parallel to the acrylic plate. All wiring was added between the STM32, breadboard, IMU, and receiver PCBs; loose wires were secured using cable ties. Lastly, the spacing between the ultrasonic receiver elements and the center was validated to be 25.0cm ± 0.1cm, and the spacing between individual receiver elements was validated to be 35.5cm ± 0.5cm

The final step of assembly was attaching the iSBL array to the Fo-SHIP end effector. The 3D-printed adapter was secured to the end effector using M3 screws; the iSBL array was secured to the adapter using additional M3 screws and nuts. All connections were tightened in a star pattern to avoid misalignments and to reduce the slop in the system. See Figure 3.24 for the fully-assembled iSBL array.

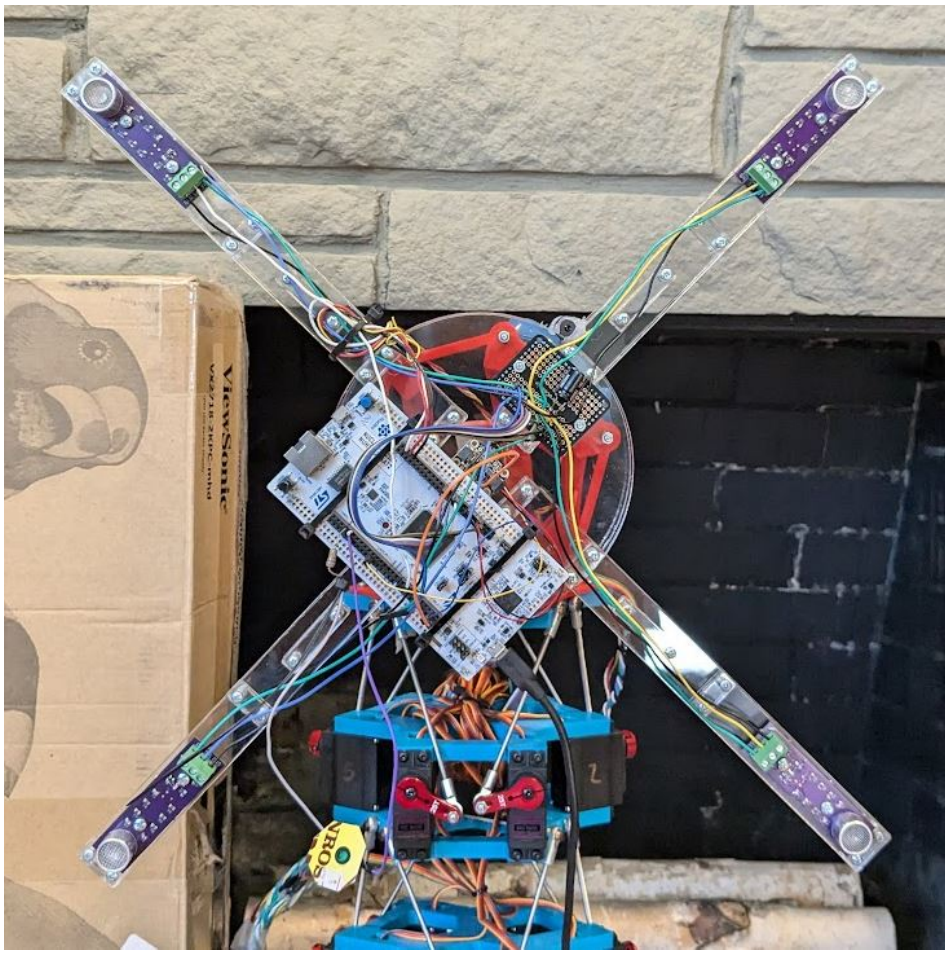


Figure 3.24: Fully-assembled iSBL array

3.6 Transmitter Design

The purpose of the transmitter system is to send a particular acoustic pulse towards the receiver array using an ultrasonic transducer. This section details the hardware design of the transmitter, followed by the software design.

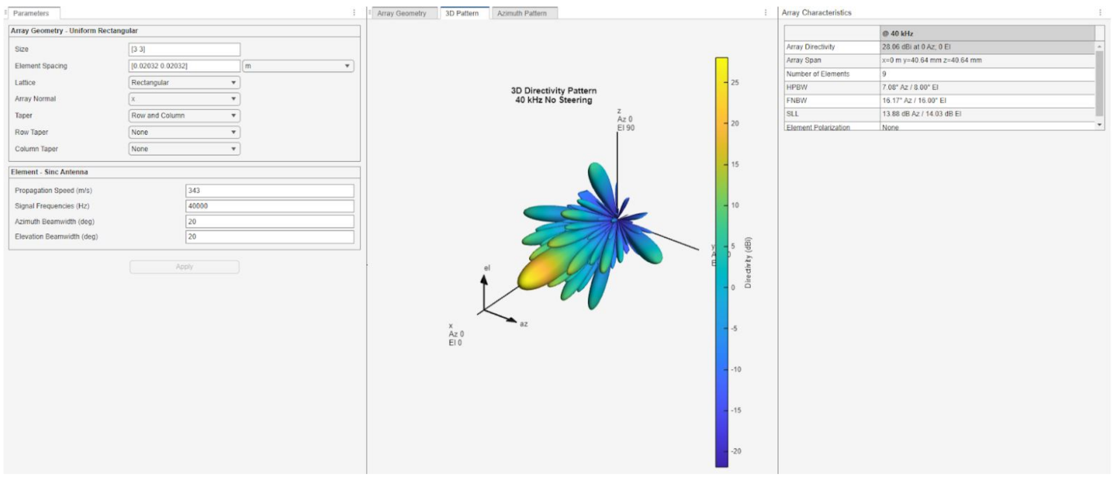
3.6.1 Transmitter Hardware Design

The hardware design of the transmitter was one of the most iterated designs in this entire thesis. The basic design involves a microcontroller for generating a particular pulse, a motor driver (or other driver IC) to control the voltage across an ultrasonic transducer, and an ultrasonic transducer (single or multiple) to transmit the sound towards the receiver array.

The microcontroller used is an STM32F411CEU6, on a development board commonly referred to as a "BlackPill" [19]. This microcontroller development board has a small form factor, is fairly low cost, and has high performance. Since the only physical connections to the microcontroller were two GPIO pins controlling the motor driver and a ground connection, the form factor was acceptable. Additionally, the microcontroller's only function would be to generate a waveform (no multitasking), so extremely high performance like the STM32H723ZG provides would not be necessary.

The motor driver went through many iterations before finalizing a design. To work with the 40kHz transducers, it needed to have a switching frequency at or above 80kHz, and needed to provide up to 24Vpp at the voltage output. Initially, spare motor driver ICs were tested and deemed unfit for the purpose: a TPIC0107B motor driver had a switching frequency of 2kHz, far below the 80kHz required; an LM386 audio amplifier had a maximum output voltage of 12Vpp, which was half the maximum of the ultrasonic transducer's range and did not produce a loud enough signal; an LM18298 motor driver had an adequate maximum output voltage, but only had a switching frequency of 25kHz; and an H-bridge constructed out of only P-type MOSFETs worked but wasted half its output power. Eventually, a spare motor driver made using optical switches by Charlie Refvem, one of the members of this thesis committee, was tested and performed fantastically. Many thanks to Charlie for loaning out the motor driver!

Lastly, the ultrasonic transducer chosen for this system is a 400ST160, manufactured by Pro-Wave Electronics Corporation [18]. This transducer has a center frequency of 40kHz and an acceptable beam angle of 55° at half-power. Two transducer configurations were tested: a single transducer, and a 3x3 array of transducers with 2.03cm spacing. The 3x3 array used a single input (all positive legs soldered together, and all negative legs soldered together); a phased array was considered, but was ultimately deemed out-of-scope. The array was simulated in Matlab (see Figure 3.25) and shows a much shallower beam angle of approximately 7.5° for half-power. Experimentally, the 3x3 array delivered the same amount of acoustic energy at 28ft as a single transmitter did at 10ft; using the inverse square law for sound propagation, the 3x3 array has approximately 7.84x the power of a single transmitter, slightly less than the ideal 9x.



Array Geometry - Uniform Rectangular

Size	[3 3]
Element Spacing	[0.02032 0.02032] m
Lattice	Rectangular
Array Normal	x
Taper	Row and Column
Row Taper	None
Column Taper	None

Element - Sinc Antenna

Propagation Speed (m/s)	343
Signal Frequencies (Hz)	40000
Azimuth Beamwidth (deg)	20
Elevation Beamwidth (deg)	20

Array Characteristics

	@ 40 kHz
Array Directivity	28.06 dBi at 0 Az; 0 El
Array Span	x=0 m y=40.64 mm z=40.64 mm
Number of Elements	9
HPBW	7.08° Az / 8.00° El
FNBW	16.17° Az / 16.00° El
SLL	13.88 dB Az / 14.03 dB El
Element Polarization	None

Figure 3.25: MATLAB simulation of 3x3 array

Figure 3.26 shows an image of the fully assembled system, with the 3x3 array connected to the output of the motor driver. Ultimately, the much shallower beam angle of the 3x3 was producing poor results with the receiver array; the transmitter would need to be a minimum of 12.5ft away to hit each receiver element with a half-power pulse. Additionally, the array caused some phasing issues with the multiple transmitter elements, which ultimately produced a slightly different signal at each receiver. Since the acoustic model assumes a single transmitter and an identical (but time-shifted) signal at each receiver, the 3x3 array would not be ideal, and the single transmitter was used for the remainder of the testing for this thesis.

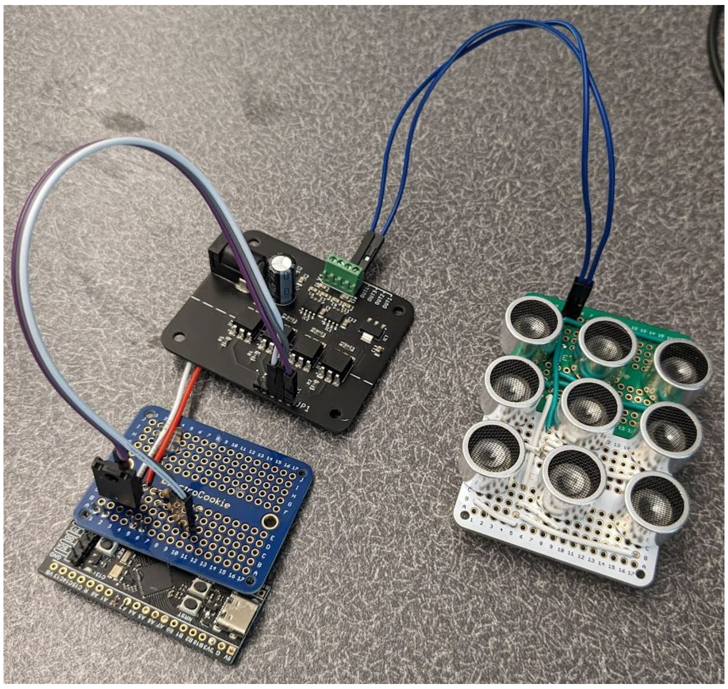


Figure 3.26: Transmitter system with 3x3 array (eventually swapped with single transmitter)

3.6.2 Transmitter Software Design

The STM32F411CEU6 was programmed using an ST-LINK/V2 from a spare Nucleo board. The code was developed in STM32CubeIDE, a software tool designed for programming STM32 microcontrollers. The code has two main functions: to generate a string of instructions for the motor driver, and to execute those instructions by toggling GPIO pins connected to the motor driver.

Originally, a planned component of this thesis was to transmit GPS data over ultrasonic audio waves; an 8-bit character would be sent over ultrasonic, encoded using on-off keying, and then decoded using a neural network [20]. However, this was unable to be implemented into the thesis in time for submission. This functionality would allow the transmission of GPS coordinates from the transmitter buoy to the underwater vehicle, and is planned for future implementation.

First, the instructions are generated. This occurs once per second. An 8-bit character is chosen and converted to its binary representation and stored in a list. The first six bits of the list are a header; they are meant to trigger the receivers to start recording.

```c
message_int ++;      // pick the next number
message_int %= 256;    // between 0 and 255 (8-bit length),
message_char = (char)message_int;    // and save it as a char

// save in the last 8 bits of message_string
for(int i = 8; i > 0; i--){
  message_string[14-i] = (message_int >> (i-1)) & 1;
}

message_string[0] = 1;  // initialize the message with a header
message_string[1] = 1;
message_string[2] = 1;
message_string[3] = 1;
message_string[4] = 0;
message_string[5] = 0;
```

The `message_string` is a list of 1s and 0s which will be transmitted using ultrasonic pulses - in a set time frame, one pulse denotes a 1 bit, and no pulse denotes a 0 bit. However, single pulses cannot be used for this case! Since the operating frequency is the same as the resonant frequency of the receivers, sending a single 40kHz wave would produce significant ringing in the transmitters. This ringing would affect data transmission rates if attempting to transmit bits over audio; additionally, sharper impulses provide the best results for time-differencing (as performed in Section 3.8), so reducing the ringing is a primary concern.

A paper by researchers in the United Kingdom details a method for reducing this ringing by implementing active damping [21]. Figure 3.27 shows two active damping waveforms and their results compared to a single transmitter pulse. Note that the second waveform produces an 80.7% decrease in the length of the received pulse.

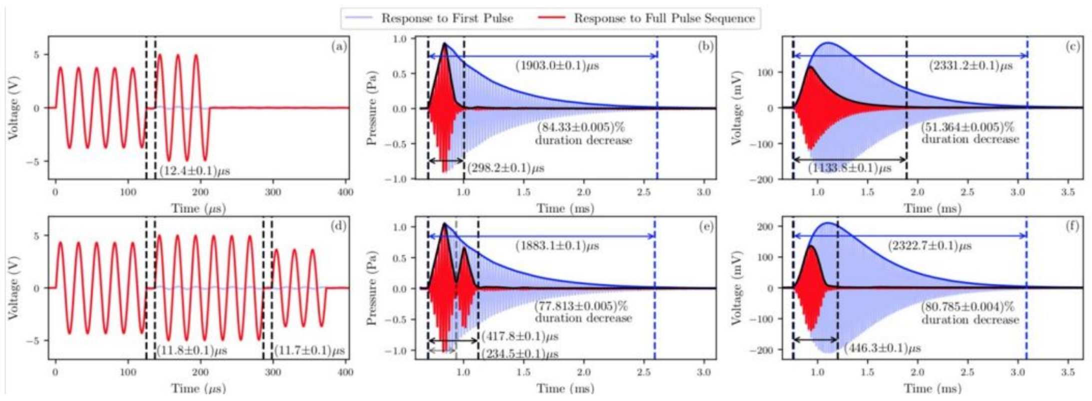


Figure 3.27: Active damping of ultrasonic receivers using engineered waves [21]

In the paper, the researchers are working with 40.01kHz audio waves. The second waveform shows a delay of 11.75us between sets of waves; these are approximately (but not perfectly) delays of half a wavelength. To simplify implementation, this thesis uses a modified waveform from the researchers' second waveform: five 40kHz waves, one half-phase delay, six 40kHz waves, one half-phase delay, and three 40kHz waves.

This modified waveform was implemented in the microcontroller code. The optical sensor motor driver takes in two inputs, A and B. A truth table for the motor driver can be seen in Table 3.1 below. The positive lead of the transducer is connected to Out1 and the negative lead is connected to Out2. By holding the same voltage across both leads, the motion of the transducer is halted; by oscillating between the second and third row of the truth table at 80kHz, a 40kHz square-wave can be generated to drive the transducer.

Table 3.1: Truth table for motor driver

Logic level, InA	Logic level, InB	Voltage, Out1	Voltage, Out2
0	0	0V	0V
0	1	+12V	0V
1	0	0V	+12V
1	1	+12V	+12V

In the code, two lists are defined: `on_a_string` provides a list of logic levels for InA, and `on_b_string` provides a list of logic levels for InB. If the GPIO pins connected to these inputs are set to the instructions in these lists at a rate of 80kHz, a square version of the modified waveform is produced. See Figure 3.28 for the waveform produced by the GPIO pins when the code is run. A sinusoidal waveform was attempted and would produce better results, but the square waveform requires a much slower update rate and produced acceptable results at the receiver. Note that trailing ones are set to allow an appropriate delay between subsequent pulses.

```
/**< list of GPIO ODR instructions to send a 1 bit over audio,
    positive input to motor driver */
uint8_t on_a_string[] = {
  1, 0,
  1, 0,
  1, 0,
  1, 0,
  1, 0,
  1,
  1, 0,
  1, 0,
  1, 0,
  1, 0,
  1, 0,
  1, 0,
  1,
  1, 0,
  1, 0,
  1, 0,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

/**< list of GPIO ODR instructions to send a 1 bit over audio,
    negative input to motor driver */
uint8_t on_b_string[] = {
  0, 1,
  0, 1,
  0, 1,
  0, 1,
  0, 1,
  1,
  0, 1,
  0, 1,
  0, 1,
  0, 1,
  0, 1,
  0, 1,
  1,
  0, 1,
  0, 1,
  0, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
```

Now, the `message_string` generated previously is combined with the above strings to produce a list of instructions. This list of instructions, `driver_string`, details what the GPIO port associated with the motor driver pins should set its output data register (ODR) to every 11.25us. For every 1 bit in `message_string`, it fills the `driver_string` with the instructions in `on_a_string` and `on_b_string`. For every 0 bit in `message_string`, it fills the `driver_string` with an equivalent number of zeros. The code assumes that InA is connected to pin B6 and InB is connected to pin B7 on the STM32F411CEU6.

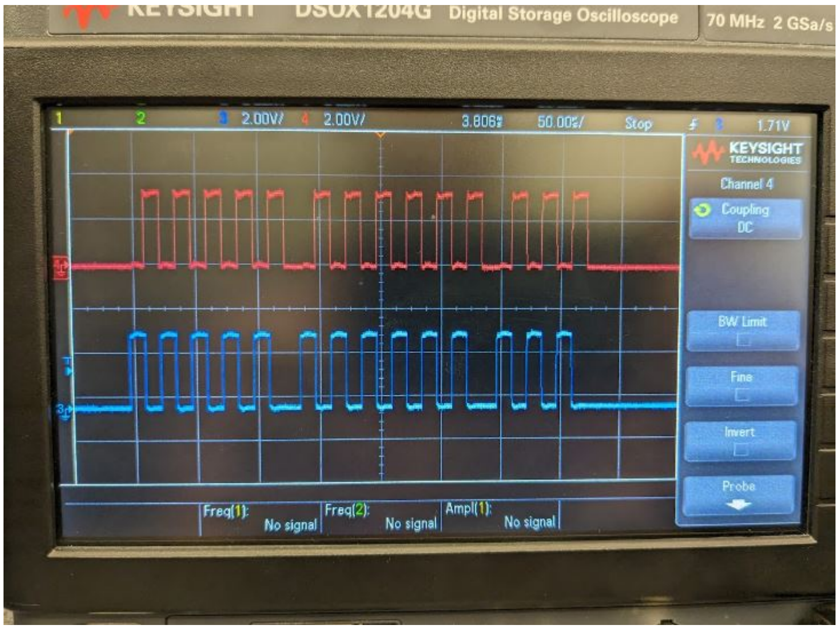


Figure 3.28: Output of GPIO pins producing square waveform, input to motor driver

```
int index = 0;
for (int i = 0; i < sizeof(message_string); i++) {
  if (message_string[i] == 1) {
    for (int j = 0; j < sizeof(on_a_string); j++) {
      driver_string[index] = (on_a_string[j] << 6) | (on_b_string[
   j] << 7);
      index++;
    }
  } else {
    for (int j = 0; j < sizeof(on_a_string); j++) {
      driver_string[index] = 0;
      index++;
    }
  }
}
```

After the message is generated, a timer is started that updates at 80kHz. Every update, the GPIO ODR is set to the next value of `driver_string`. Since each message bit takes 40 cycles of an 80kHz timer to send, the effective data rate of the transmission is approximately 2kHz. This means that in an ideal environment, data can be sent from the transmitter to the receiver at 2kb/s. Figure 3.29 shows the 8-bit number 241, represented in binary as 11110001, as received by the iSBL receiver array.

```
if (htim->Instance == TIM4){
  GPIOB->ODR = driver_string[instruction_counter];
  instruction_counter++;
  if (instruction_counter >= DRIVER_STRING_LEN){
    instruction_counter = 0;
    HAL_TIM_Base_Stop(&htim4);
  }
}
```

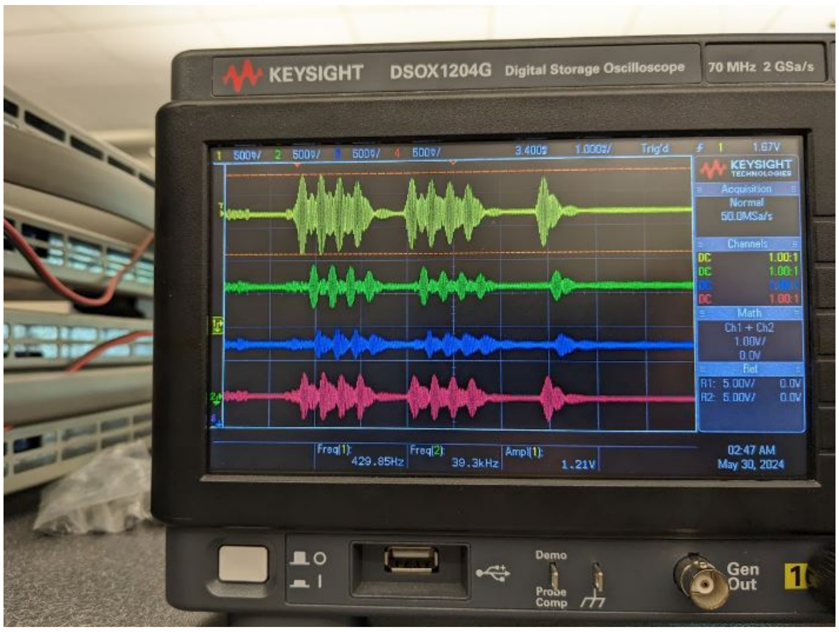


Figure 3.29: 8-bit number 241 (11110001) transmitted over ultrasonic

This process repeats once per second. Though the transmitter sends a message with multiple individual 40kHz pulses and 14 data bits, the transmission sent every second will be referred to as "a single acoustic pulse" for the remainder of this paper.

3.7 Recording an Audio Pulse

This section is the first of three detailing the acoustic position estimate code running on the STM32H723ZG. This first section details the overall finite-state machine diagram for the system, how the ADC is configured for MultiMode, the two different recording states (triggering and full recording), and how a "good" pulse is determined.

3.7.1 Finite-State Machine

Figure 3.30 shows the finite-state machine representation of this system. There are five main states: initialization, triggering, full recording, recording validation, and data processing.

In the initialization state, three timers are started: TIM13 is used to count the time between Madgwick filter updates (see Section **??**), TIM14 is used to count the time between acoustic position updates (see Section **??**), and TIM3 is used to trigger ADC conversions (see Section 3.7.2). In this state, the first ADC conversion is started and the system moves into the triggering state.

In the triggering state, a small number of samples are recorded from each microphone (64, in this implementation). These samples are tested to see if they exceed a threshold defined by `TRIG_THRES`. If more than one microphone exceeds the threshold, then the system moves to the full recording state. Otherwise, the

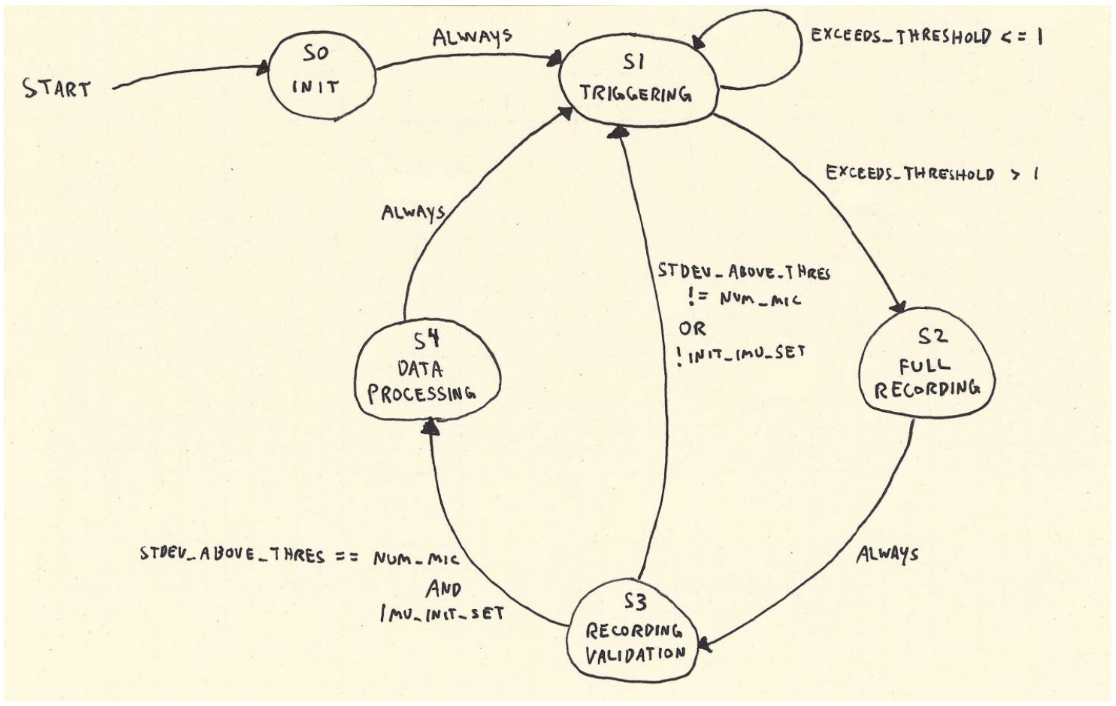


Figure 3.30: Finite-state machine diagram for acoustic positioning system

system re-enters the triggering state. This state is crucial for detecting the header bits of an acoustic pulse and triggering the full recording process; without this step, a total of 2048 samples per microphone would need to be checked constantly. See Section 3.7.3 for details on the triggering state.

In the full recording state, 2048 samples are recorded from each microphone (roughly 2.4ms of recording time, which is a good portion of a full 7ms acoustic pulse). The system does not update the IMU during this state to prevent any signal interference or cross-talk. After the recording buffer is full, the system transitions to the recording validation state.

In the recording validation state, the mean and standard deviation of each microphone's signal is calculated. If all the standard deviations exceed `TRIG_THRES` and the IMU has initialized, then the system moves to the data processing state. If either of these conditions are not met, the system restarts at the triggering state. See Section 3.7.4 for details on the full recording and recording validation states.

In the data processing state, a position estimate is formed. First, the time shift between the microphone signals is calculated using FFT cross-correlation (see Section 3.8). Next, the position of each microphone in the acoustic array is rotated by the current orientation of the platform (see Section 3.9.1) Then, the position of the transmitter relative to the receiver array is estimated using Hooke-Jeeves search using an acoustic propagation model (see Section 3.9.2). This relative position estimate is combined with the known position of the transmitter to give a position estimate of the receiver array (see Section 3.9.3). Finally, the acoustic position estimate and the dead reckoning position estimate are combined using Kalman filters (see Chapter **??**).

In the background, linear acceleration, angular velocity, and magnetic field strength data are being recorded from the IMU at 208Hz. This data is then converted into an orientation estimate using a Madgwick filter, and into a position estimate using dead reckoning. See Chapter **??** for details.

3.7.2 ADC Configuration

The sampling rate of the ADC determines the resolution of the time shift calculation. The time shift between two signals sampled at 80kHz can only be determined to within 12.5us; assuming audio signals in water with a speed of sound of 1500 m/s [15], this means that the distance from each microphone to the transmitter can only be discerned to within 9.4mm. This shift is enough to introduce significant error into the position estimation algorithm, so increasing the sampling rate of the ADC is a primary concern.

The ADC has multiple resolutions to choose from (8-bit, 12-bit, 16-bit, etc.). After testing, 12-bit resolution was found to provide enough resolution for the filtered audio signals with a resolution of approximately 0.81mV per bit. A single 12-bit ADC conversion takes approximately 12.5 cycles to convert on the STM32 [13], and higher resolutions take even longer.

Due to the input impedance of the active band-pass filter PCBs, additional ADC cycles were needed to stabilize the voltage output of the filter PCBs. Different sampling times were tested, and 8.5 cycles per conversion was found to provide a stable signal without requiring excess sampling time. Running at the maximum ADC clock speed of 50MHz and combining the standard conversion time of 12.5 cycles with the additional sampling time of 8.5 cycles, it takes approximately 0.42us to complete a single ADC conversion for one microphone.

Initially, only one ADC on the microcontroller was used for all four microphones. When the timer TIM3 updated, it would trigger a series of four subsequent conversions, one from each microphone. See Figure 3.31 for a timing diagram of these conversions. Since all four conversions happened subsequently, the maximum theoretical sampling rate for the ADC was approximately 595kHz. In practice, 500kHz was the maximum possible sampling rate - anything above this rate produced incomplete conversions for the final channel.

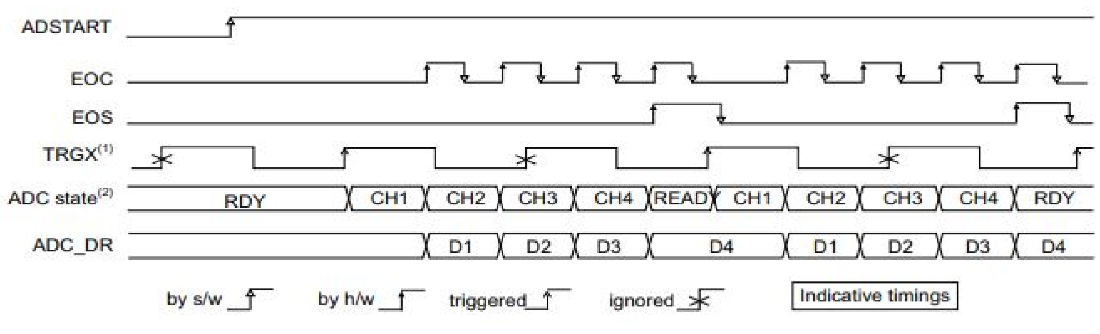


Figure 3.31: Timing diagram for four subsequent ADC conversions [13]

However, using a configuration called MultiMode (also called ”dual ADC” mode) allows the sampling rate to be doubled. In this configuration, one ADC (ADC1) is designated as the primary ADC, and a different ADC (ADC2) is designated as the secondary ADC. When the primary ADC is triggered, it then automatically triggers the second ADC to start recording 1.5 cycles later [13]. Both the primary and secondary ADC use two channels; ADC1 reads from microphones 0 and 2, and ADC2 reads from microphones 1 and 3. See Figure 3.32 for a timing diagram of this MultiMode configuration. Note that the figure shows two ADCs running in MultiMode with four channels each; the thesis implementation uses two ADCs with only two channels each.

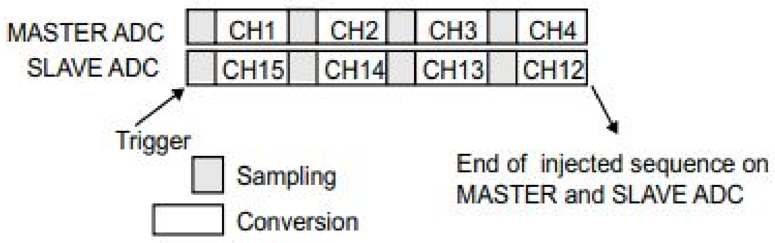


Figure 3.32: Timing diagram for MultiMode ADC conversions [13]

Since MultiMode uses two ADCs running simultaneously (with a 1.5 cycle delay), the maximum theoretical sampling rate for the primary ADC is approximately 1.15MHz. In practice, 851.4kHz was the maximum possible sampling rate - anything above this rate produced incomplete conversions for the final two channels.

After conversion, the ADC readings must be saved in a data buffer. Manually writing them to the buffer would significantly impact the maximum allowed sampling rate, so direct memory access (DMA) is used. A DMA stream is configured to write the ADC conversions to a memory buffer; however, using MultiMode requires special handling of these conversions. In normal modes, the 12-bit ADC conversions are stored as `uint16_t` values. MultiMode saves simultaneous readings as `uint32_t` values, with the first conversion (ADC1) stored in the first 16 bits of the `uint32_t` value and the second (ADC2) stored in the last 16 bits of the `uint32_t` value. Two macros were created to extract the individual ADC readings from these values. The code snippet below shows these two macros, as well as the initialization sequence for the ADCs in the initialization state.

```
#define adc1conv(data) ((data) & 0xFFFF)
#define adc2conv(data) ((data >> 16) & 0xFFFF)
...
// start DMA from ADCs, writing TRIG_BUF_LEN samples to trig_buf (
    triggering state)
HAL_ADC_Start(&hadc2);
HAL_ADCEx_MultiModeStart_DMA(&hadc1, (uint32_t*)trig_buf,
    TRIG_BUF_LEN * NUM_MIC / 2);
// start timer that controls ADCs
HAL_TIM_Base_Start_IT(&htim3);
```

3.7.3 Triggering State

The triggering state is used to decide if an acoustic pulse is incoming. In this state, the ADC records 64 samples from each microphone at 851.4kHz and saves their values to the buffer `trig_buf`. Once the ADC conversion is finished and all readings are saved to the buffer, the recordings from each individual microphone are extracted from the buffer and the mean of each signal is calculated.

```c
float mic0_mean = 0, mic1_mean = 0, mic2_mean = 0, mic3_mean = 0;
for (int i = 0; i < TRIG_BUF_LEN * NUM_MIC/2; i += NUM_MIC/2) {
  mic0_mean += (float)adc1conv(trig_buf[i]);
  mic1_mean += (float)adc2conv(trig_buf[i]);
  mic2_mean += (float)adc1conv(trig_buf[i+1]);
  mic3_mean += (float)adc2conv(trig_buf[i+1]);
}
mic0_mean /= (float)TRIG_BUF_LEN;
mic1_mean /= (float)TRIG_BUF_LEN;
mic2_mean /= (float)TRIG_BUF_LEN;
mic3_mean /= (float)TRIG_BUF_LEN;
```

Next, the program checks if any of the microphones have recorded signals with sufficient variance. If any five subsequent values in a microphone's signal exceed `TRIG_THRES`, then that microphone has registered a significant signal. The code loops through every set of five values in each microphone's signal and determines if any of the sets contain values all greater than `TRIG_THRES`. If they do, then the flag `exceedsThresholdmicN` is set to 1 for that microphone. This provides a robust method for determining when to record a full signal. For a 40kHz wave sampled at 851.4kHz, there are approximately 21 samples per full waveform; checking for five samples in a row detects the peaks of these waveforms when recording only a small number of samples.

```c
#define NUM_THRES_ROW 5
...
int exceedsThresholdmic0 = 0;
int exceedsThresholdmic1 = 0;
int exceedsThresholdmic2 = 0;
int exceedsThresholdmic3 = 0;
int loopLen = TRIG_BUF_LEN - NUM_THRES_ROW;
for (int i = 0; i < loopLen; i += 1) {
  int thres_count_mic0 = 0;
  int thres_count_mic1 = 0;
  int thres_count_mic2 = 0;
  int thres_count_mic3 = 0;

  // check if the next NUM_THRES_ROWS values exceed TRIG_THRES
  for (int t = 0; t < NUM_THRES_ROW; t++) {
    thres_count_mic0 += fabsf((float)adc1conv(trig_buf[i*NUM_MIC/2
    + t*NUM_MIC/2 + 0]) - mic0_mean) > TRIG_THRES;
    thres_count_mic1 += fabsf((float)adc2conv(trig_buf[i*NUM_MIC/2
    + t*NUM_MIC/2 + 0]) - mic1_mean) > TRIG_THRES;
    thres_count_mic2 += fabsf((float)adc1conv(trig_buf[i*NUM_MIC/2
    + t*NUM_MIC/2 + 1]) - mic2_mean) > TRIG_THRES;
    thres_count_mic3 += fabsf((float)adc2conv(trig_buf[i*NUM_MIC/2
    + t*NUM_MIC/2 + 1]) - mic3_mean) > TRIG_THRES;
  }

  if (thres_count_mic0 == NUM_THRES_ROW) exceedsThresholdmic0 = 1;
  if (thres_count_mic1 == NUM_THRES_ROW) exceedsThresholdmic1 = 1;
  if (thres_count_mic2 == NUM_THRES_ROW) exceedsThresholdmic2 = 1;
  if (thres_count_mic3 == NUM_THRES_ROW) exceedsThresholdmic3 = 1;

  // If all mics have been exceeded, we can break the loop
  if (exceedsThresholdmic0 && exceedsThresholdmic1 &&
   exceedsThresholdmic2 && exceedsThresholdmic3) {
    break;
  }
}
```

After this, the total number of microphone signals that met the threshold criteria is summed. If more than one microphone met the threshold criteria, then the system transitions to the full recording state. Otherwise, the triggering state restarts and the above process repeats. In this state, each microphone has an associated LED for debugging; if the LED for a microphone lights up, then that microphone's signal exceeded the threshold. This was crucial for ensuring the transmitter was pointing directly at the acoustic array during the data collection in Chapter ??.

```c
int exceedsThreshold = exceedsThresholdmic0 + exceedsThresholdmic1
     + exceedsThresholdmic2 + exceedsThresholdmic3;

// if at least two microphones exceeded the threshold, move to the
     full recording state
if (exceedsThreshold > 1) {
  state = 2;                  // transition to full recording state
  recording = 1;              // prevent interrupting ADC recording
  HAL_ADC_Start(&hadc2);      // start the secondary ADC
  HAL_ADCEx_MultiModeStart_DMA(&hadc1, (uint32_t*)adc_buf,
   ADC_BUF_LEN * NUM_MIC / 2); // start the primary ADC in DMA
   mode

  // otherwise, restart the triggering process
} else {
  HAL_ADC_Start(&hadc2);      // start the secondary ADC
  HAL_ADCEx_MultiModeStart_DMA(&hadc1, (uint32_t*)trig_buf,
   TRIG_BUF_LEN * NUM_MIC / 2); // start the primary ADC in DMA
   mode
}
```

3.7.4 Full Recording and Recording Validation State

Not much happens during the full recording state besides ADC conversions. In this state, the STM32 halts reading from the IMU and updating the Madgwick filter. Even though the ADC conversions happen in the background and are written to a memory buffer using DMA, it is imperative that the ADC sampling occurs at the maximum sampling rate, so any potential sources of noise or resource use are halted.

Once the ADC conversion is finished and 2048 samples are recorded from each microphone, the system transitions to the recording validation state. This state is similar to the triggering state: its objective is to determine if the recording has enough variance to accurately perform a time shift calculation. Unlike the triggering state, a different method is used for determining the quality of the signal. The vast length of the full recording signals means that the standard deviation can be used as a good metric for the signal variance; it is assumed, due to the filtering of non-40kHz signals and amplification of 40kHz signals, that a signal with a sufficiently-large standard deviation contains the acoustic pulse from the transmitter. This was verified experimentally during testing in Chapter **??**.

First, the mean and standard deviation of each signal is calculated. If the standard deviation of all four signals exceeds `TRIG_THRES` and the IMU has been initialized, then the recording is considered sufficient for further data processing and the system transitions to the data processing state. If any of the signals fails to exceed the threshold, then the system returns to the triggering state; if this is the case, then the ADC buffers are reset.

```c
// set recording flag to 0 because the ADC has stopped recording
recording = 0;

// stop the ADC trigger timer
HAL_TIM_Base_Stop_IT(&htim3);

// variables for calculating average and standard deviation for
    each microphone
mic0_avg = 0, mic1_avg = 0, mic2_avg = 0, mic3_avg = 0;
mic0_var = 0, mic1_var = 0, mic2_var = 0, mic3_var = 0;
mic0_stdev = 0, mic1_stdev = 0, mic2_stdev = 0, mic3_stdev = 0;

// calculate average of each mic signal
for (int i = 0; i < ADC_BUF_LEN * NUM_MIC / 2; i += NUM_MIC / 2) {
  mic0_avg += (float)adc1conv(adc_buf[i]);
  mic1_avg += (float)adc1conv(adc_buf[i+1]);
  mic2_avg += (float)adc2conv(adc_buf[i]);
  mic3_avg += (float)adc2conv(adc_buf[i+1]);
}
mic0_avg /= (float)ADC_BUF_LEN;
mic1_avg /= (float)ADC_BUF_LEN;
mic2_avg /= (float)ADC_BUF_LEN;
mic3_avg /= (float)ADC_BUF_LEN;

// calculate variance of each mic
for (int i = 0; i < ADC_BUF_LEN * NUM_MIC / 2; i += NUM_MIC / 2) {
  mic0_var += powf((float)adc1conv(adc_buf[i]) - mic0_avg, 2);
  mic1_var += powf((float)adc1conv(adc_buf[i+1]) - mic1_avg, 2);
  mic2_var += powf((float)adc2conv(adc_buf[i]) - mic2_avg, 2);
  mic3_var += powf((float)adc2conv(adc_buf[i+1]) - mic3_avg, 2);
}
mic0_var /= (float)(ADC_BUF_LEN - 1);
mic1_var /= (float)(ADC_BUF_LEN - 1);
mic2_var /= (float)(ADC_BUF_LEN - 1);
mic3_var /= (float)(ADC_BUF_LEN - 1);

// calculate standard deviation of each mic
mic0_stdev = sqrtf(mic0_var);
mic1_stdev = sqrtf(mic1_var);
mic2_stdev = sqrtf(mic2_var);
mic3_stdev = sqrtf(mic3_var);
```

```

// check if each microphone exceeds the threshold for "good" data
int stdev_above_thres = 0;
stdev_above_thres += mic0_stdev > (TRIG_THRES);
stdev_above_thres += mic1_stdev > (TRIG_THRES);
stdev_above_thres += mic2_stdev > (TRIG_THRES);
stdev_above_thres += mic3_stdev > (TRIG_THRES);

// if the signal is good and the initial IMU angle has been set (
    remove yaw), then
// move to the data processing state
if ((stdev_above_thres >= NUM_MIC) && imu_init_set){
  state = 4;
}
else{
  // reset all ADC buffers
  memset(adc_buf, 0, sizeof(adc_buf));
  memset(trig_buf, 0, sizeof(trig_buf));
  ...
  // return to the trigger state
  state = 1;

  // restart the DMA from ADCs and the timer that controls the
    ADCs
  HAL_ADC_Start(&hadc2);
  HAL_ADCEx_MultiModeStart_DMA(&hadc1, (uint32_t*)trig_buf,
    TRIG_BUF_LEN * NUM_MIC / 2);
  HAL_TIM_Base_Start_IT(&htim3);
}
```

Figure 3.33 shows an acoustic pulse recorded from four microphones on a Keysight oscilloscope. Figure 3.34 shows an acoustic pulse (with a slightly different receiver orientation) recorded using the STM32 and ADCs. In that figure, the ADC readings have been normalized, as described in Section 3.8.2. The ADC readings tend to be slightly noisier than the oscilloscope readings, but are sufficient for calculating time shifts.

3.8 Calculating Time Shift

Once a pulse has been recorded from all four microphones, the time shift between microphone recordings is calculated. Cross-correlations are a standard method for computing the time shift between two similar signals, but this method can be quite computationally-taxing for large signals; for this reason, the fast Fourier transform (FFT) cross-correlation method is used. This takes the computation time from $O(n^2)$ time to $O(n \times log(n))$ time [22]. This section describes the theory of the FFT cross-correlation, then shows its implementation on the microcontroller.

3.8.1 FFT Cross-Correlation Theory

Before the FFT cross-correlation can be understood, the basics of the convolution operation must be discussed. Cross-correlation and convolution are effectively the same operation, but with one of the signals flipped horizontally.

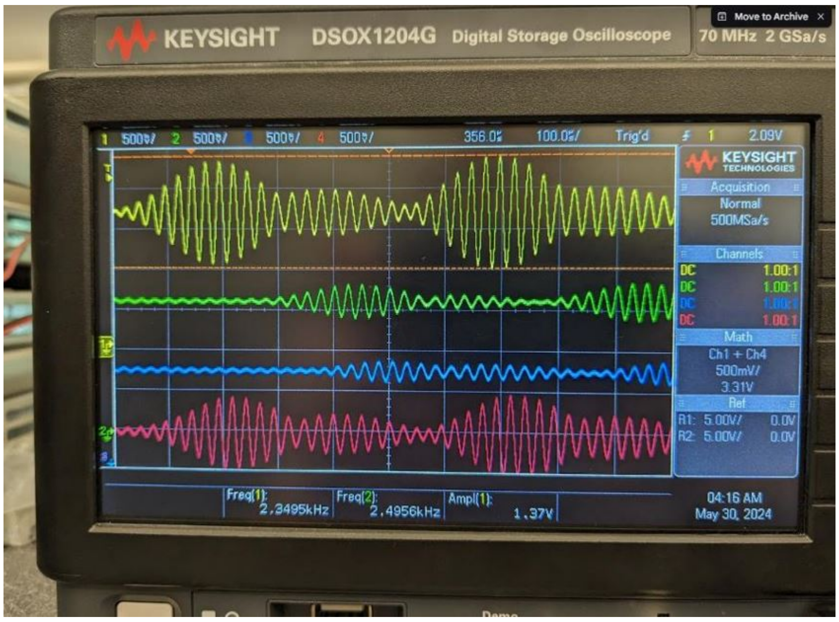


Figure 3.33: Microphone recordings on Keysight oscilloscope

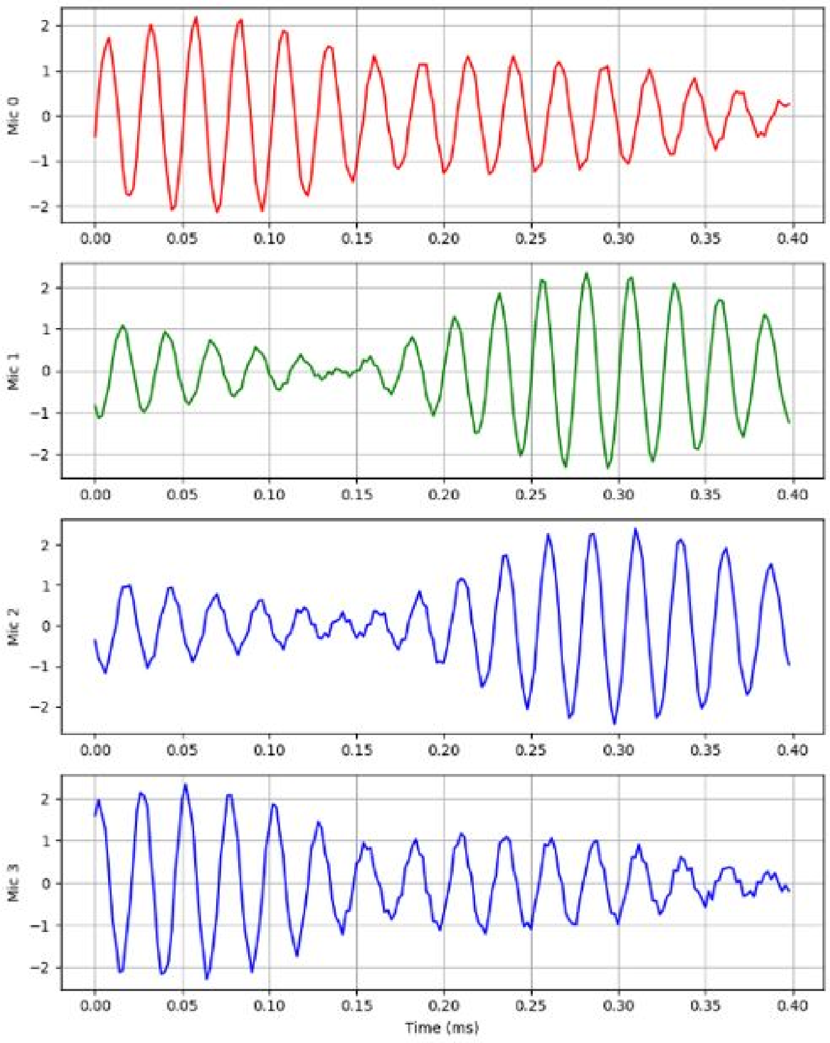


Figure 3.34: Microphone recordings on STM32 ADCs, slightly different receiver orientation

The convolution is a mathematical operation that takes two signals and produces a third signal [22]. The mathematical notation for a convolution of discrete signals is:

$$x[n] * h[n] = y[n] \tag{3.1}$$

Figure 3.35 shows an example convolution, while Figure 3.36 shows the components that make up the output of the convolution. A convolution operation can be thought of as flipping the second signal horizontally and then sliding it over the first signal, summing the area underneath the combination of both signals. Figure 3.37 gives a wonderful visualization of the process, and the GIF on the Wikipedia page for cross-correlation is a fantastic animation showing how the cross-correlation of two signals is computed.

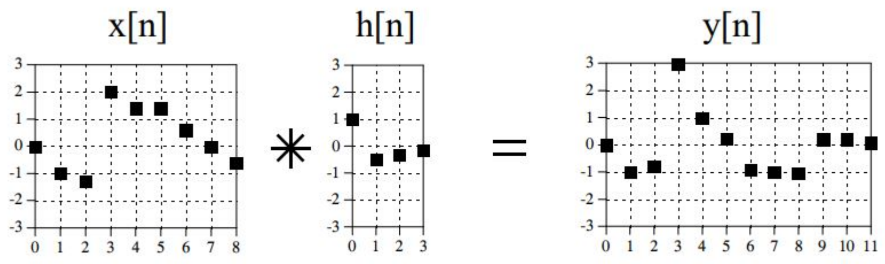


Figure 3.35: Convolution operation for two discrete signals [22]

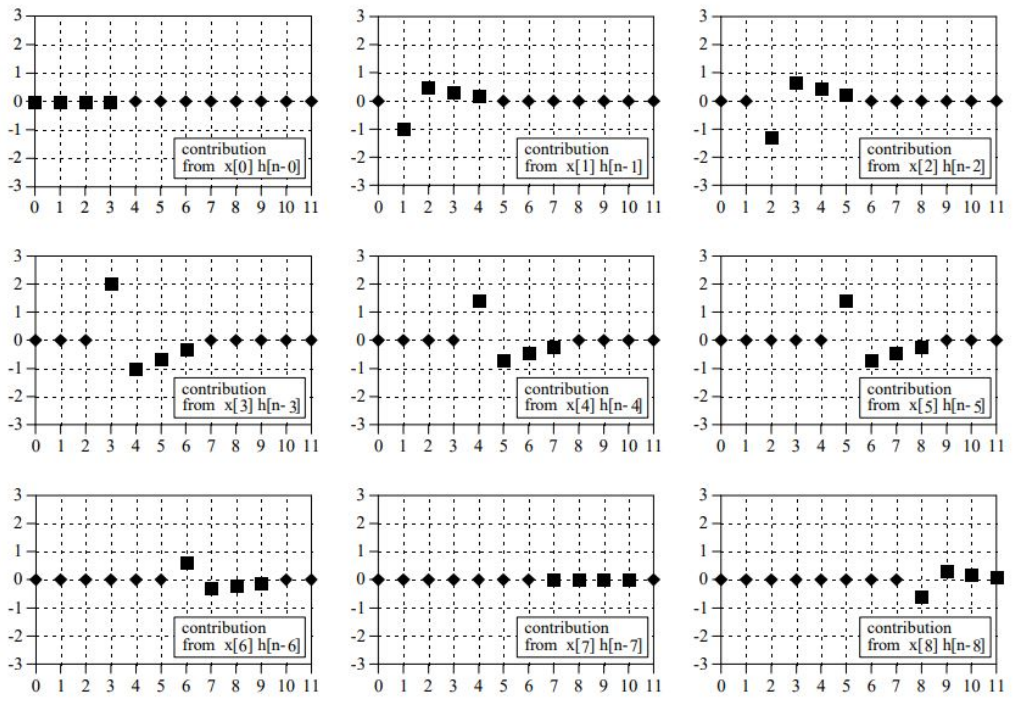


Figure 3.36: Convolution operation for two discrete signals, step-by-step process [22]

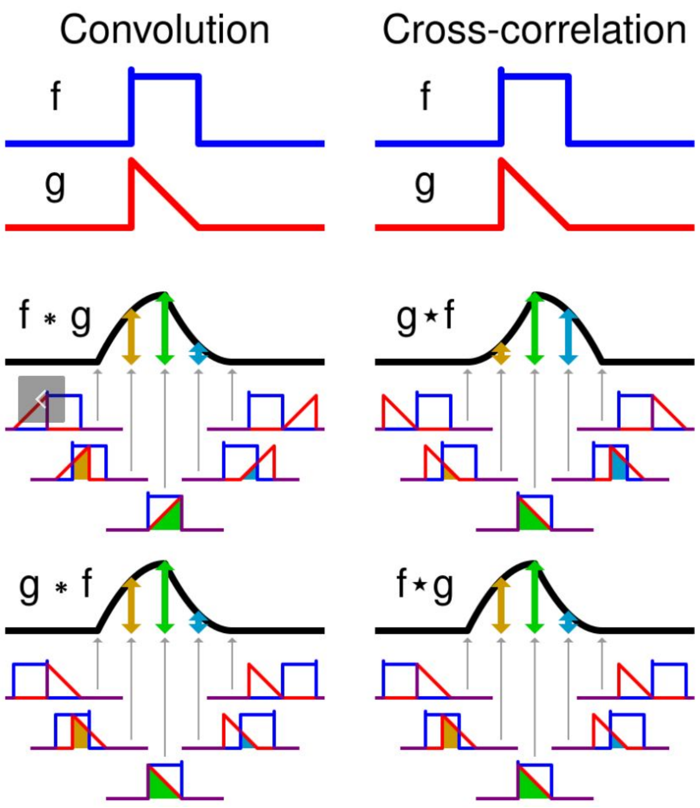


Figure 3.37: Convolution and cross-correlation operation for continuous signals [23]

The convolution operation can be used to find the time shift between two similar signals. By first flipping the second signal horizontally (to negate the flip from the convolution operation), the two signals can be convolved to effectively slide them over each other. This process of flipping the second signal prior to the convolution is called "cross-correlation." In the output of the cross-correlation, the largest positive value corresponds to the shift between the signals [22]. This should make sense graphically; when the two signals fully overlap, the peaks of both align and the area under the curve shared between the signals is at a maximum. Figure 3.38 shows this for two triangular pulses, with the shift between the two being correctly estimated as -0.5 (signal 2 lags signal 1 by 0.5 time units).

When computing the convolution, it is important to add "padding zeros" to the end of both signals. Without this step, the convolution operation is unable to fully "slide" the two signals over each other; it would instead wrap around to the beginning of the signal to complete the operation, which would give an incorrect result. For two signals of equal length n, a total of n padded zeros should be added to each signal [22].

The standard convolution operation is very computationally-expensive for long signals, with a convolution of two signals of equal length n taking $O(n^2)$ time. However, convolution in the time domain corresponds to multiplication in the frequency domain, which is a much more efficient operation [22]. This changes the computation time for those same signals to $O(n \times log(n))$ time. For signals

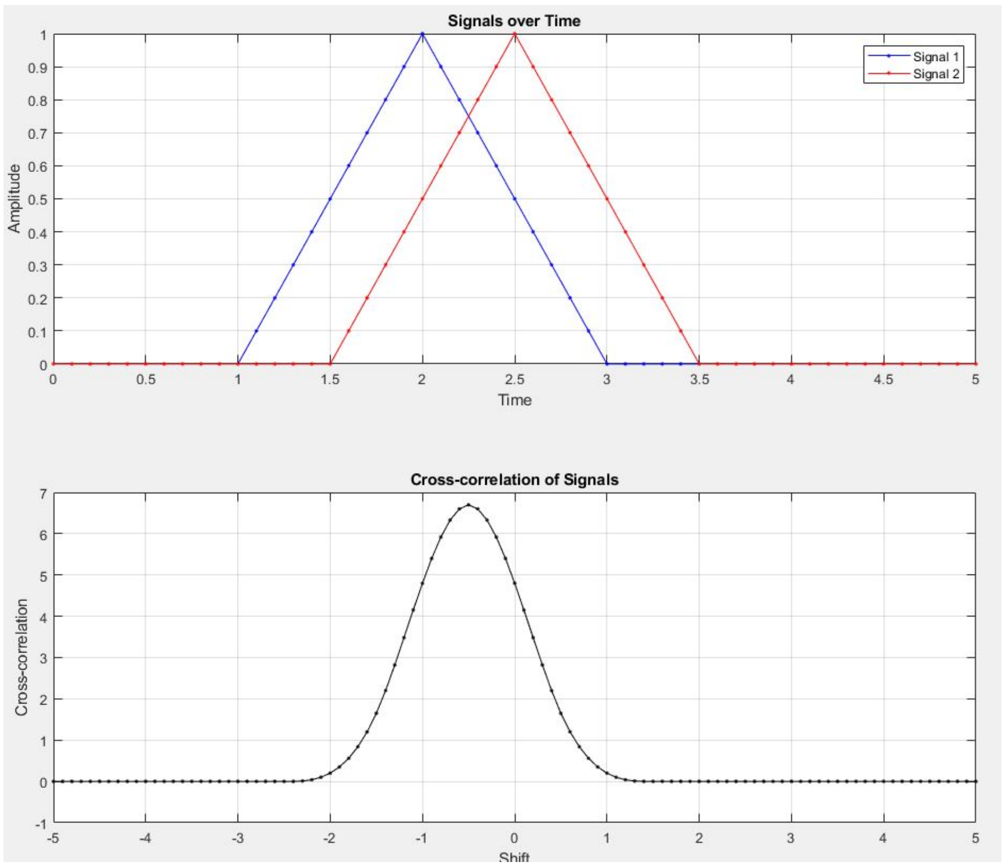


Figure 3.38: Cross-correlation of two triangular pulses

of length 4096 (as this implementation uses, more details in next subsection), moving the operation to the frequency domain decreases the number of arithmetic operations required by approximately 99.9%!

The name of this operation is the FFT convolution, or high-speed convolution. Figure 3.39 shows a graphical explanation for the operation. Both signals are converted from the time domain to the frequency domain using the FFT operation, multiplied in the frequency domain, and then the product is converted back to the time domain using the inverse FFT operation. For more details on the FFT and inverse FFT, see Chapter 12 of *The Scientist and Engineer's Guide to Digital Signal Processing* [22].

Complex multiplication is required for the frequency domain; signals in that domain have both a real and imaginary component. Complex multiplication is carried out as shown below [22]:

$$(a + bj)(c + dj) = (ac - bd) + j(bc + ad) \tag{3.2}$$

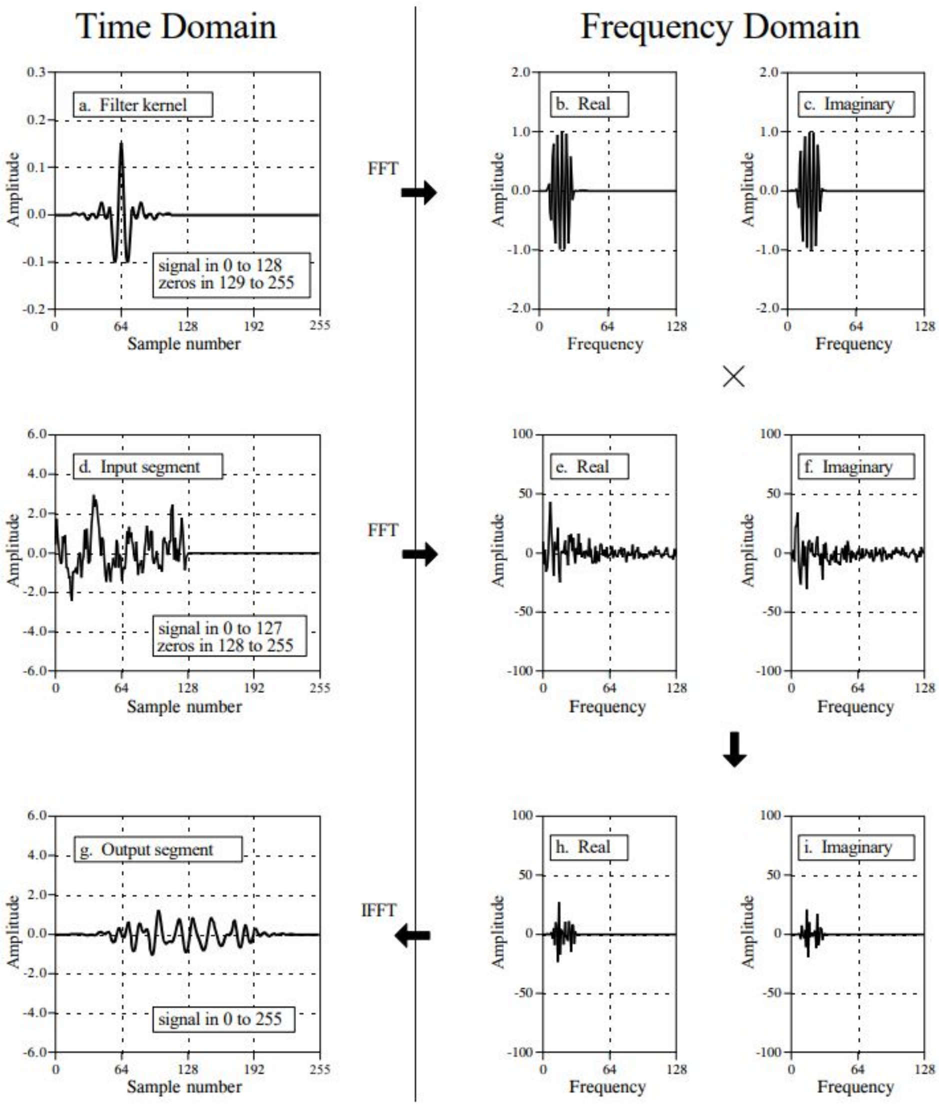


Figure 3.39: FFT convolution visual guide [22]

Finally, to perform FFT cross-correlation instead of FFT convolution, two approaches can be taken. First, the second signal can be flipped horizontally. Second, the complex conjugate of the second signal's FFT transform can be taken. These operations are equivalent; one is in the time domain, and one is in the frequency domain. For this implementation, the complex conjugate operation is chosen, and its definition can be seen below [22]:

$$conj(a + bj) = (a - bj) \tag{3.3}$$

3.8.2 FFT Cross-Correlation Implementation

The STM32 implementation makes heavy use of the `arm_math.h` digital signal processing library [24]. This library implements all of the operations described in the previous section and many more. It is written for ARM Cortex M0, M3, M4, and M7 cores; the STM32H723ZG uses a Cortex M7 core. This subsection will walk through the code implementation of the FFT cross-correlation, which is used in the function `calculate_time_shift()`.

Once the system transitions to the data processing state, the first step is to normalize the signals from each of the four microphones. Then, the function `calculate_time_shift()` is run on each unique pair of microphone signals; all signals are compared to microphone 0 for simplicity. In future implementations, designing the program to compare the signals to the "best" signal (highest standard deviation, least noise) would be ideal.

```c
// buffers for holding normalized microphone buffers
float32_t mic0_buf[ADC_BUF_LEN];
float32_t mic1_buf[ADC_BUF_LEN];
float32_t mic2_buf[ADC_BUF_LEN];
float32_t mic3_buf[ADC_BUF_LEN];

// de-interleave the ADC values and normalize the data
int index = 0;
for (int i = 0; i < ADC_BUF_LEN * NUM_MIC / 2; i += NUM_MIC / 2) {
  mic0_buf[index] = (((float)adc1conv(adc_buf[i]) - mic0_avg) /
   mic0_stdev);
  mic1_buf[index] = (((float)adc1conv(adc_buf[i+1]) - mic1_avg) /
   mic1_stdev);
  mic2_buf[index] = (((float)adc2conv(adc_buf[i]) - mic2_avg) /
   mic2_stdev);
  mic3_buf[index] = (((float)adc2conv(adc_buf[i+1]) - mic3_avg) /
   mic3_stdev);
  index += 1;
}

// calculate time-shifts between mic0 and all other microphones
measured_time_shifts[0] = calculate_time_shift(mic0_buf,mic1_buf);
measured_time_shifts[1] = calculate_time_shift(mic0_buf,mic2_buf);
measured_time_shifts[2] = calculate_time_shift(mic0_buf,mic3_buf);
```

Separately, a handler for the FFT operations used in `arm_math.h` is initialized in the initialization state of the program. This handler defines the length of buffers used in these operations and must be a power of two. More details for the initialization can be found in the `arm_math.h` documentation [24].

```c
arm_rfft_fast_init_f32(&fftHandler, FFT_BUF_LEN);
```

The time shift function takes in pointers to two data buffers as inputs. These buffers contain the normalized data from two microphone signals. First, the microphone signals are moved into temporary buffers; padded zeros equal to the length of each signal are added. Then, the real FFT of each signal is computed (since both signals only contain real values, not complex). For memory efficiency, some buffers are reused.

```c
#define ADC_BUF_LEN 2048
#define FFT_BUF_LEN ADC_BUF_LEN*2
...
float32_t calculate_time_shift(float32_t *micA_buf, float32_t *
    micB_buf) {
  // load buffers
  float32_t fftBufIn[FFT_BUF_LEN];    /**< input to FFT */
  float32_t fftBuf1Out[FFT_BUF_LEN];  /**< output of first FFT */
  float32_t fftBuf2Out[FFT_BUF_LEN];  /**< output of second FFT */
  float32_t ifftBufIn[FFT_BUF_LEN];   /**< input to inverse FFT */
  float32_t ifftRealOut[FFT_BUF_LEN]; /**< output of inv FFT */

  // reset buffers to 0
  memset(fftBufIn, 0, sizeof(fftBufIn));
  memset(fftBuf1Out, 0, sizeof(fftBuf1Out));
  memset(fftBuf2Out, 0, sizeof(fftBuf2Out));
  memset(ifftBufIn, 0, sizeof(ifftBufIn));
  memset(ifftRealOut, 0, sizeof(ifftRealOut));

  // load micA values into fftBufIn (remaining values zeros for
   zero-padding)
  memcpy(fftBufIn, micA_buf, ADC_BUF_LEN * sizeof(float32_t));

  // perform real fft on zero-padded micA values
  arm_rfft_fast_f32(&fftHandler, fftBufIn, fftBuf1Out, 0);

  // reset fftBufIn
  memset(fftBufIn, 0, sizeof(fftBufIn));

  // load micB values into fftBufIn (remaining values zeros for
   zero-padding)
  memcpy(fftBufIn, micB_buf, ADC_BUF_LEN * sizeof(float32_t));

  // perform real fft on zero-padded micB values, put result into
   temporary buf
  arm_rfft_fast_f32(&fftHandler, fftBufIn, ifftBufIn, 0);
```

Next, the complex conjugate of the second signal's FFT is taken. The FFT of the first signal and the complex conjugate of the second's FFT are then multiplied using complex multiplication. Finally, the inverse real FFT of the product is taken (to return only real values, not complex) and stored in a buffer.

```c
// calculate complex conjugate of micB real fft output (required
  for cross-correlation)
arm_cmplx_conj_f32(ifftBufIn, fftBuf2Out, FFT_BUF_LEN);

// reset temporary buf
memset(ifftBufIn, 0, sizeof(ifftBufIn));

// complex multiplication of the two fft output bufs
arm_cmplx_mult_cmplx_f32(fftBuf1Out, fftBuf2Out, ifftBufIn,
 FFT_BUF_LEN);

// perform inverse real fft on multiplication result
arm_rfft_fast_f32(&fftHandler, ifftBufIn, ifftRealOut, 1);
```

The results in the inverse FFT output buffer are the output of the FFT cross-correlation operation. To find the most likely time shift between the signals, the index of the largest positive value is found.

```c
// find argmax(ifft_output) to find time shift between signals
int maxIndex = 0;
float32_t maxVal = 0.0f;
for (int i = 0; i < FFT_BUF_LEN; i++) {
  float32_t curVal = (ifftRealOut[i]);
  if (curVal > maxVal) {
    maxVal = curVal;
    maxIndex = i;
  }
}
```

The x-axis of the cross-correlation results are not straightforward to understand. Figure 3.40 shows how the `arm_math.h` implementation outputs its cross-correlation result of two signals. Compare this to Figure 3.38, which shows the shift on the x-axis instead of the output index.

To convert the maximum index of the output to a time shift, all values need to be shifted by half the number of output samples. Then, the index can be converted to a time shift in seconds using the sampling rate of the ADCs. The code below accomplishes this:

```c
// shift is maxIndex if maxIndex < 0.5*fft_len, otherwise is
 maxIndex - fft_len
float32_t adj_index = (maxI ndex < FFT_BUF_LEN/2) ? (float)(
 maxIndex) : (float)(-FFT_BUF_LEN + maxIndex);

// convert shift index to seconds (275MHz / 323 ADC clock)
float32_t shift = adj_index / (275000000 / 323);
```

Finally, the temporary buffers are reset, and the function returns `shift`. All three shifts are stored in a list called `measured_time_shifts[]`.

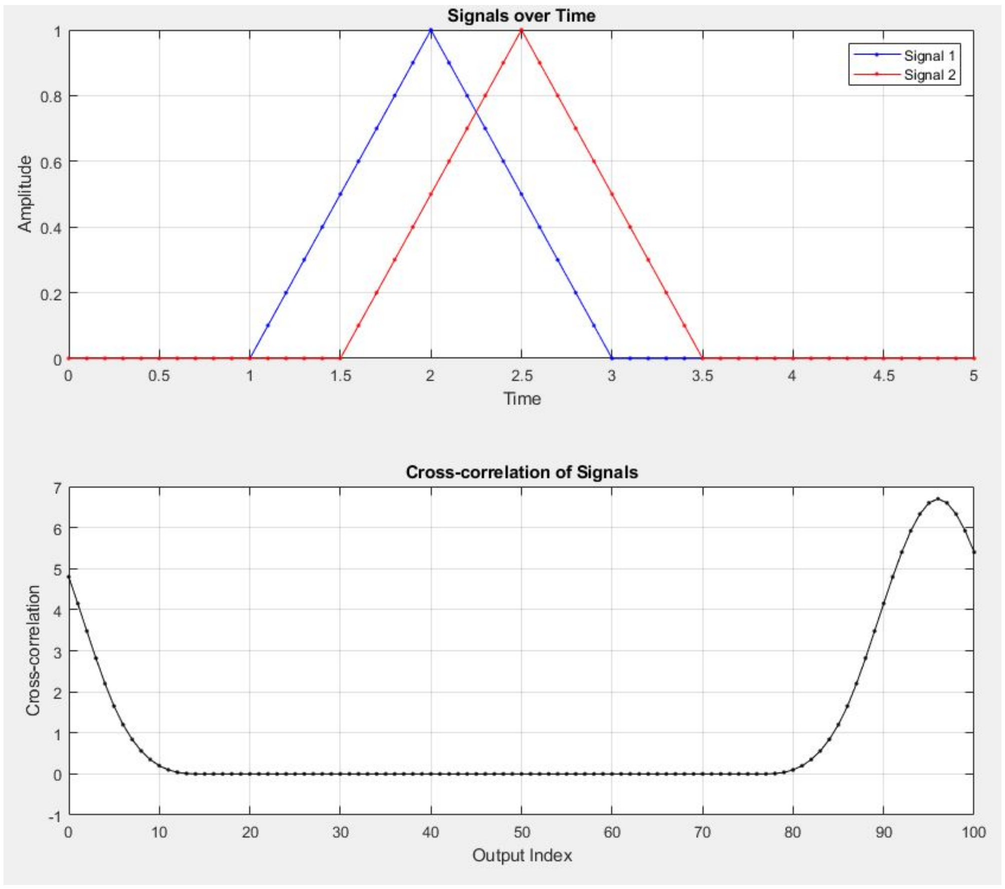


Figure 3.40: Cross-correlation of two triangular pulses, STM32 implementation

3.9 Acoustic Position Estimation

After the time shifts between microphones have been calculated, the position of the receiver array's center relative to the transmitter can be estimated. This section focuses on the position estimation workflow: rotating the microphones' 3D positions by the current orientation of the array, running Hooke-Jeeves Search to get a relative position estimate, and incorporating the transmitter position to get the position estimate of the receiver array.

3.9.1 Rotating the Microphone Array

The first step for position estimation is rotating the microphone array by its current orientation relative to the test frame. Section ?? covers the specifics of how that orientation is obtained; for this section, it is assumed that the IMU and Madgwick filter have initialized and the quaternion representation of the array's orientation is available in floats `q0, q1, q2,` and `q3`. Section ?? also explains what a quaternion is; for brevity, it is an alternate way of expressing the rotation of one coordinate frame relative to another, whose elements fulfill the relationship in Equations 3.4 and 3.5.

$$q = q_0 + iq_1 + jq_2 + kq_3 \tag{3.4}$$

$$i^2 = j^2 = k^2 = ijk = -1 \tag{3.5}$$

First, the initial yaw component of the orientation is removed from the current orientation estimate. This is covered in more detail in Section ??, but for testing and data collection purposes, the position estimates are collected in a different coordinate frame than the global frame. The "global frame" refers to the NED (positive X points North, positive Y points East, positive Z points down) frame relative to the Earth's magnetic and gravitational fields. For data collection, it was much easier to use the local frame of the Fo-SHIP: positive X points forward, positive Y points to the right, and positive Z points down. These are relative to the zero set point of the Fo-SHIP, which is on flat and level ground as measured by accelerometers.

To convert the readings to the local frame of the Fo-SHIP (henceforth known as the "test frame"), the yaw component of the initial orientation estimate must be removed. This is achieved in three steps: creating a yaw-only quaternion, rotating the current orientation quaternion by the yaw-only quaternion, and normalizing the result.

```
// extract the current orientation of the platform
// note: q0-3 are calculated in the Madgwick filter
Quaternion quat_raw = {q0, q1, q2, q3};

// create a quaternion to undo initial yaw rotation
Quaternion yaw_compensation = create_yaw_quaternion(init_yaw);

// apply the yaw compensation to the current quaternion
Quaternion quat = multiply_quaternions(yaw_compensation,quat_raw);
```

```c

// normalize the result to ensure it's a valid rotation
normalize_quaternion(&quat);
```

The majority of the functions in this subsection are sourced from D. Rose's article "Rotation Quaternions, and How to Use Them" [25]. The first function, `create_yaw_quaternion()`, takes in an initial yaw in radians and forms a yaw-only quaternion. It is a partial version of a currently-unused function `EulerAnglesToQuaternion()`, which is available in the code if needed. The initial yaw given in the code has already been negated during the IMU initialization, so this rotation represents the cancellation of the initial yaw.

```c
Quaternion create_yaw_quaternion(float yaw) {
  Quaternion q;
  q.w = cos(yaw / 2.0f);
  q.x = 0.0f;
  q.y = 0.0f;
  q.z = sin(yaw / 2.0f);
  return q;
}
```

Next, the two quaternions are multiplied. Multiplying quaternion A by quaternion B is the same as rotating a coordinate system by quaternion A first, followed by rotating it by quaternion B. For this implementation, it removes the initial yaw element. Quaternion multiplication is associative but non-commutative, so care must be taken in the order of the inputs. The multiplication function is derived in D. Rose's article [25].

```c
Quaternion multiply_quaternions(Quaternion q1, Quaternion q2) {
  Quaternion result;
  result.w = q1.w*q2.w - q1.x*q2.x - q1.y*q2.y - q1.z*q2.z;
  result.x = q1.w*q2.x + q1.x*q2.w + q1.y*q2.z - q1.z*q2.y;
  result.y = q1.w*q2.y - q1.x*q2.z + q1.y*q2.w + q1.z*q2.x;
  result.z = q1.w*q2.z + q1.x*q2.y - q1.y*q2.x + q1.z*q2.w;
  return result;
}
```

Lastly, the product is normalized. The multiplication of two quaternions should produce a quaternion that obeys Equations 3.4 and 3.5, but floating point operations can cause small errors to accumulate. Normalization ensures that the quaternion is valid.

```c
void normalize_quaternion(Quaternion* q) {
  float magnitude = sqrt(q->w*q->w + q->x*q->x + q->y*q->y + q->z*
    q->z);
  q->w /= magnitude;
  q->x /= magnitude;
  q->y /= magnitude;
  q->z /= magnitude;
}
```

Now that the quaternion represents the orientation of the array relative to the test frame, the 3D coordinates of each microphone need to be rotated. The need for this operation is demonstrated graphically in Figure 3.41. Without rotating the microphone array's coordinates, the position estimate provided in the following subsections would only give the position of the transmitter normal to the face of the array! Ultimately, the position of the transmitter in the test frame is desired, so the orientation must be accounted for.

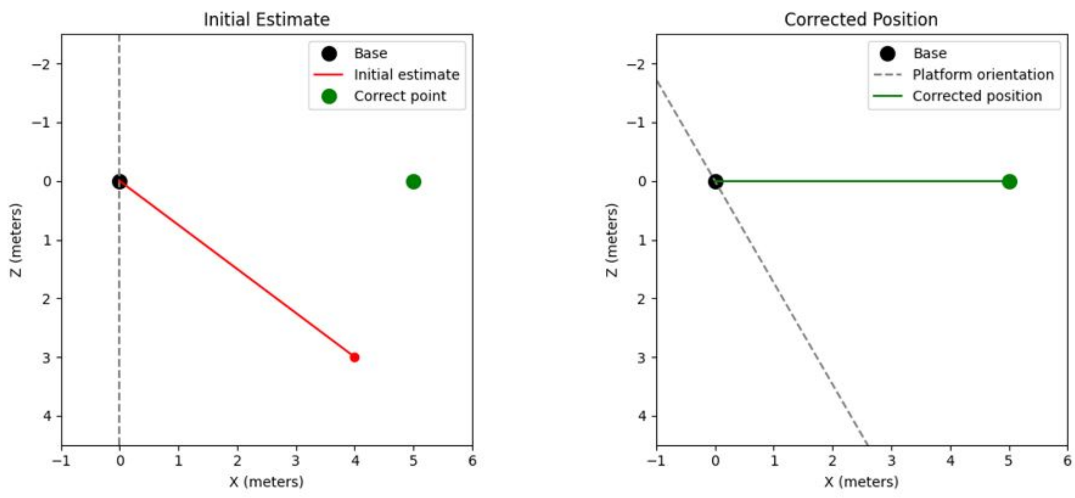


Figure 3.41: Orientation correction demonstration, 2D case

The 3D coordinate of each microphone's center is stored in a custom struct, `MicArray_t`. It makes use of another custom struct, `Point3D`.

```c
typedef struct {
  float x, y, z;
} Point3D;

typedef struct {
  uint32_t num_mics;
  Point3D base_points[];  // Flexible array member
} MicArray_t;
```

In the initialization stage, the un-rotated `MicArray_t` struct is formed. The system uses a coordinate system parallel to the test frame, and assumes that the center of the array is the origin (as opposed to the test frame, which uses the center of the Fo-SHIP's bottom platform as the origin).

```c
float32_t baseline = 0.250f
...
// initialize the mic array base points
MicArray_t* original_mic_array = createMicArray(NUM_MIC);
calculate_base_points(original_mic_array, baseline);
...
MicArray_t* createMicArray(uint32_t num_mics) {
  MicArray_t* mic_array = (MicArray_t*)malloc(sizeof(MicArray_t) +
     num_mics * sizeof(Point3D));
  if (mic_array != NULL) {
    mic_array->num_mics = num_mics;
  }
  return mic_array;
}
```

```c
void calculate_base_points(MicArray_t* mic_array, float baseline)
  {
  // convert the baseline to a YZ spacing factor
  const float factor = baseline / 1.414f;

  // set coordinates for each microphone
  mic_array->base_points[0] = (Point3D){0.0f, -factor, -factor};
  mic_array->base_points[1] = (Point3D){0.0f, -factor,  factor};
  mic_array->base_points[2] = (Point3D){0.0f,  factor,  factor};
  mic_array->base_points[3] = (Point3D){0.0f,  factor, -factor};

  // if there are more than 4 microphones, set their coordinates
   to (0,0,0)
  // note: this implementation only uses 4, so this is just for
   error catching
  for (uint32_t i = 4; i < mic_array->num_mics; i++) {
    mic_array->base_points[i] = (Point3D){0.0f, 0.0f, 0.0f};
  }
}
```

Once in the data processing state, the original `MicArray_t` is rotated by the yaw-compensated quaternion. Essentially, the function iterates through each 3D point in the `MicArray_t`, and rotates it by the supplied quaternion.

```c
MicArray_t* rotated_mic_array = rotateMicArray(original_mic_array,
     quat);
...
MicArray_t* rotateMicArray(MicArray_t* original_mic_array,
    Quaternion quat) {
  if (original_mic_array == NULL) {
    return NULL;
  }

  MicArray_t* rotated_mic_array = createMicArray(
   original_mic_array->num_mics);
  if (rotated_mic_array == NULL) {
    return NULL;
  }


  for (uint32_t i = 0; i < original_mic_array->num_mics; i++) {
    rotated_mic_array->base_points[i] = rotatePoint(
   original_mic_array->base_points[i], quat);
  }

  return rotated_mic_array;
}
```

Rotating a point by a quaternion is quite similar to multiplying two quaternions together. The point, with coordinates (X, Y, Z), is placed into its own quaternion $v = 0 + iX + jY + kZ$. The point can be rotated by quaternion q by performing the multiplications in Equation 3.6, where q^{-1} is the inverse of the quaternion [25].

$$v' = qvq^{-1} \tag{3.6}$$

```
Point3D rotatePoint(Point3D point, Quaternion q) {
  Point3D rotated;
  float x = point.x, y = point.y, z = point.z;

  // First multiplication: q * v
  float tw = -q.x*x - q.y*y - q.z*z;
  float tx =  q.w*x + q.y*z - q.z*y;
  float ty =  q.w*y - q.x*z + q.z*x;
  float tz =  q.w*z + q.x*y - q.y*x;

  // Second multiplication: (q * v) * q^-1
  // Note: for a unit quaternion, q^-1 = conjugate of q = (q.w, -q
   .x, -q.y, -q.z)
  rotated.x = tw*(-q.x) + tx*q.w + ty*(-q.z) - tz*(-q.y);
  rotated.y = tw*(-q.y) - tx*(-q.z) + ty*q.w + tz*(-q.x);
  rotated.z = tw*(-q.z) + tx*(-q.y) - ty*(-q.x) + tz*q.w;

  return rotated;
}
```

After these steps, the microphones will have been successfully rotated by the current orientation of the platform relative to the test frame, and the Hooke-Jeeves search algorithm can be run.

3.9.2 Hooke-Jeeves Search

The majority of this part of the thesis implementation is based on the work from the Underwater Communication and Navigation Laboratory. Their GitHub repository, UCNLNav, contains a Matlab implementation of a variable-baseline time-difference-of-arrival acoustic position estimator [12], which is precisely what this thesis is implementing. Significant modifications were made to their implementation, but their work forms the backbone of the acoustic position estimation system.

The Hooke-Jeeves search (HJS) algorithm is the keystone of the acoustic position estimation system. Also known as "pattern search," it is an algorithm for finding the minimum of a function in an arbitrary number of dimensions [26]. It was designed by Robert Hooke and T. A. Jeeves when they were working at Westinghouse Research Laboratories [27]. The most similar algorithm is gradient descent, an algorithm that finds the minimum of a function using information about the derivatives / gradient of the function. HJS is beneficial in that it does not require information about the gradient; this will be useful for future implementations where complex ocean acoustics models are incorporated into the sound propagation model and the gradient cannot be easily derived.

The HJS algorithm for **n** dimensions can be seen below in Algorithm 1. A GIF on the Wikipedia page for pattern search provides a fantastic visualization of how HJS works.

Essentially, the algorithm starts with an initial position estimate, an initial spacing, and a scaling factor. The algorithm requires some function that takes in a position in **n** dimensions and returns a scalar value. Along each dimension, points within ± one spacing unit of the initial position estimate are tested using the residual function. Whichever direction returns the smallest residual (or largest, if trying to maximize a function) is the new direction of motion. The algorithm will keep moving in that direction using a step size of one spacing unit until the residual function begins to increase. At that point, the spacing size is divided by the scaling factor (usually 2), and a new direction is chosen. This process repeats until a set number of iterations have taken place, the spacing size has reached some minimum threshold, or the residual has reached some minimum threshold [26]. HJS and other pattern search methods are proven to converge [28], but may get stuck in local minima.

For this implementation, two choices are considered: should a 2D or 3D HJS algorithm be run? While the position estimate and receiver positions exist in three dimensions, short baselines tend to cause issues when running 3D HJS. The UCNLNav Matlab implementation was tested with the baselines of the physical iSBL array and simulated results for a transmitter approximately 7 meters away. The original code runs 3D HJS but was modified to run HJS in 2D, with results locked to a single plane parallel to the XY plane. A small amount of noise was added to measurements in order to mimic real-world environments.

Multiple planes, spaced 0.1m apart, were tested with the 2D HJS algorithm. Figure 3.42 shows the results of this testing. The star represents the true transmitter location, while the triangles represent the three best 2D HJS results, and the dots represent the HJS results for each plane. If the algorithm were able to determine the 3D position using the acoustic data alone, it would be expected that the three triangles would overlap at the star. However, the graph shows that this is not the case. In fact, 3D HJS was run in addition to the 2D test, and its position estimate is shown with the square.

The final residual of all planes between $z = -2m$ and $z = -10m$ were very close to one another, each within approximately 5% of the lowest value. It can be seen that all minimum points lie on the vector between the center of the receiver array and the true transmitter location. The 3D HJS algorithm has a difficult time determining where upon the vector the true location is, but the square does indeed lie upon the vector. It should be noted that the Matlab simulation in Figure 3.42 is flipped vertically from this thesis's implementation.

Thankfully, this system is designed for underwater use, and one particular sensor can make up for the short baseline's shortcomings: a pressure (depth) sensor. These sensors measure the depth of an underwater vehicle by converting water pressure readings into depth measurements. Relatively inexpensive and commercially-available sensors such as Blue Robotics' Bar30 pressure sensor are capable of accurate depth measurement within 2m at up to 300m deep [29]. For the underwater implementation, the depth measurement can determine the z-coordinate of the transmitter, while the HJS algorithm determines the x- and y-coordinates.

Algorithm 1 Hooke-Jeeves Search Algorithm in **n** Dimensions

```
Require: InitialPosition, MaxIterations, MinResidual, MinSpacing, ScaleFactor, InitialSpacing
Ensure: OptimizedPosition
 1: $position \leftarrow InitialPosition_{\mathbf{n}\times 1}$
 2: $newPosition \leftarrow InitialPosition_{\mathbf{n}\times 1}$
 3: $residual \leftarrow \infty$
 4: $spacing \leftarrow InitialSpacing \times ScaleFactor$
 5: $iteration \leftarrow 0$
 6: $newDirectionNeeded \leftarrow false$
 7: $direction \leftarrow \mathbf{0}_{\mathbf{n}\times 1}$
 8: while $iteration < MaxIterations$ and $residual > MinResidual$ and $spacing > MinSpacing$ do
 9:     if $newDirectionNeeded$ then
10:         $spacing \leftarrow spacing/ScaleFactor$
11:         for each dimension d in n do
12:             $direction \leftarrow \mathbf{0}_{\mathbf{n}\times 1}$
13:             $direction(\mathbf{d}) \leftarrow 1$
14:             $testPosition \leftarrow position + direction \times spacing$
15:             $residualList(2\mathbf{d}) \leftarrow \text{CalculateResidual}(testPosition)$
16:             $testPosition \leftarrow position - direction \times spacing$
17:             $residualList(2\mathbf{d}+1) \leftarrow \text{CalculateResidual}(testPosition)$
18:         end for
19:         if $\min(residualList) < residual$ then
20:             $direction \leftarrow \text{argmin}_{direction}(residualList)$
21:             $residual \leftarrow \min(residualList)$
22:             $position \leftarrow position + direction \times spacing$
23:             $newDirectionNeeded \leftarrow false$
24:         else
25:             $newDirectionNeeded \leftarrow true$
26:         end if
27:     else
28:         $prevResidual \leftarrow residual$
29:         $newPosition \leftarrow position + direction \times spacing$
30:         $residual \leftarrow \text{CalculateResidual}(newPosition)$
31:         if $residual < prevResidual$ then
32:             $position \leftarrow newPosition$
33:         else
34:             $residual \leftarrow prevResidual$
35:             $newDirectionNeeded \leftarrow true$
36:         end if
37:     end if
38:     $iteration \leftarrow iteration + 1$
39: end while
        return $position$
```

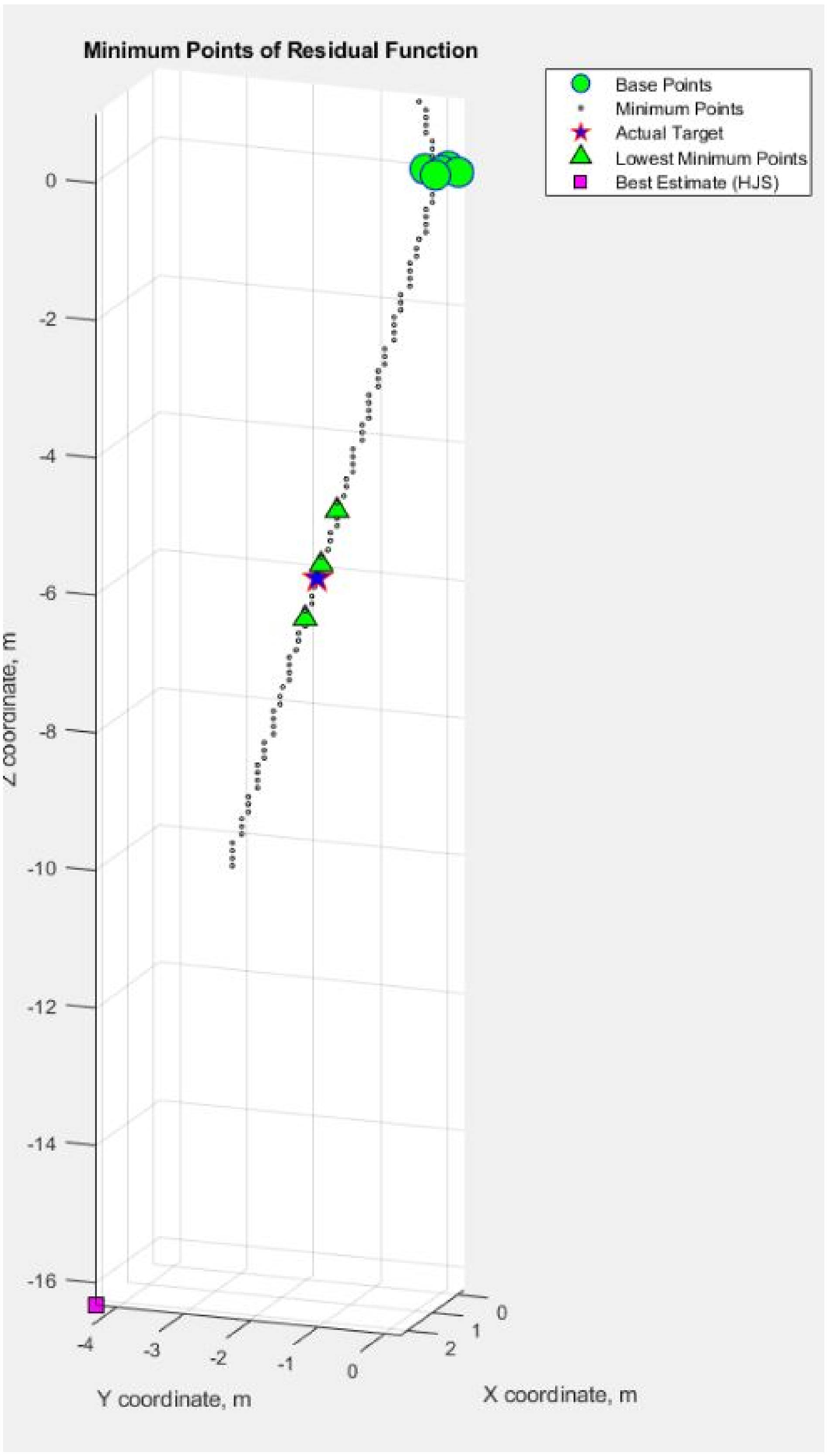


Figure 3.42: Matlab HJS results, search run on each 2D plane

For the above-water implementation and testing, the HJS algorithm was run in 2D with a simulated depth measurement providing the depth coordinate. Since the testing is performed horizontally (see Section **??** for more details), the x-coordinate represents the depth of the above-water system. The measurement was simulated, since pressure sensors provide no position information in air; this was accomplished by combining the true x-coordinate of the transmitter with the true x-coordinate of the iSBL array and adding random noise. The true position of the transmitter is known prior to testing; in the full underwater implementation, the position would be given via GPS coordinates. The true x-coordinate of the iSBL array's center is sent from the Fo-SHIP's ESP32 to the iSBL array's STM32 as described in Section **??**. Both of these values are crucial for determining the true "depth" of the system.

The initial position estimate is set with the depth measurement as the initial x-coordinate, and with zeros for the y- and z-coordinates. Then, the 2D HJS function is called. It takes in the 3D coordinates of each rotated microphone (see Section 3.9.1), the initial position estimate, the measured time shifts from each microphone (see Section 3.8.2), and the exit condition parameters below:

- A maximum of 1000 iterations (generally only takes 60-70 iterations)
- A minimum residual of 1e-13 m^2 (generally returns with 1e-9 m^2)
- A minimum spacing of 1e-6 m (generally the condition that causes termination)

If any of the exit conditions are met, then the HJS terminates and returns its current position estimate. The parameters were determined experimentally; the minimum spacing tends to be the most important, as a spacing of below 0.001mm not providing much additional accuracy. In fact, this parameter could be raised higher without affecting the position estimation accuracy by much. For the full underwater implementation, these parameters would need to be re-tuned.

```c
// the true position of the transmitter in meters
float true_x = 3.880f;
float true_y = 0.0f;
float true_z = -0.682f;

// define an initial guess for the transmitter's XYZ position
// note: noise in the depth measurement is simulated here using
   randn()
Point3D init_pos_est;
init_pos_est.x = true_x - fossl_x + randn(0.0f, 0.005f);
init_pos_est.y = 0;
init_pos_est.z = 0;

// perform hooke-jeeves search to minimize the residual function
raw_isbl_pos_est = hooke_jeeves_search_2d(rotated_mic_array,
   init_pos_est, measured_time_shifts, 1000, 1e-13, 1e-6, 2.0f);
```

The STM32 implementation of 2D HJS can be seen below in its entirety. It follows Algorithm 1 very closely, testing different positions along the y- and z-axes.

```c
Point3D hooke_jeeves_search_2d(MicArray_t* mic_array, Point3D
   init_pos_est, float32_t measured_time_shifts[], int max_iter,
   float32_t min_residual, float32_t min_spacing, float32_t
   scale_factor) {
```

```
Point3D pos_est = init_pos_est;
Point3D new_pos_est = init_pos_est;
float32_t residual = 1e9; // meters^2
float32_t residual_arr[4];
float32_t prev_residual = 1e10;
float32_t spacing = 2; // meters
int new_dir_index = 0;
int new_dir_flag = 0;
int iter = 0;
int dy = 0;     /**< direction modifier for y-axis */
int dz = 0;     /**< direction modifier for z-axis */

while ((iter < max_iter) && (residual > min_residual) && (spacing
   > min_spacing)){

  // if previous move gave a worse residual, choose new direction
  if (new_dir_flag){

    // every time a new direction is chosen, spacing is reduced
    spacing /= scale_factor;

    // test plus/minus spacing in each dimension, save residuals
    int d_idx = 0;
    for (int dy_ = -1; dy_ <= 1; dy_ += 2){
      new_pos_est.y = pos_est.y + (dy_) * spacing;
      new_pos_est.z = pos_est.z;
      residual_arr[d_idx] = calculate_residual(mic_array,
   new_pos_est, measured_time_shifts);
      d_idx ++;
    }
    for (int dz_ = -1; dz_ <= 1; dz_ += 2){
      new_pos_est.y = pos_est.y;
      new_pos_est.z = pos_est.z + (dz_) * spacing;
      residual_arr[d_idx] = calculate_residual(mic_array,
   new_pos_est, measured_time_shifts);
      d_idx ++;
    }

    // find direction that gives smallest residual
    new_dir_index = arg_min(residual_arr, d_idx);

    // if that direction has a smaller residual than the previous
   residual, move in that direction
    if (residual_arr[new_dir_index] < residual){
      residual = residual_arr[new_dir_index];

      // determine what the new direction is
      if (new_dir_index < 2) {
        dy = (new_dir_index % 2) * 2 - 1;
        dz = 0;
      } else {
        dy = 0;
        dz = ((new_dir_index - 2) % 2) * 2 - 1;
      }

      // calculate new position based on that direction
      new_pos_est.y = pos_est.y + (dy) * spacing;
      new_pos_est.z = pos_est.z + (dz) * spacing;

      // save as new best position estimate
      pos_est = new_pos_est;

      // start moving in that direction until improvement stops
      new_dir_flag = 0;
    }

    // otherwise, repeat above process
    else{
      new_dir_flag = 1;
    }
  }

  // otherwise, continue in same direction
  else{
    // calculate residual for next position
    prev_residual = residual;
    new_pos_est.y = pos_est.y + (dy) * spacing;
    new_pos_est.z = pos_est.z + (dz) * spacing;
    residual = calculate_residual(mic_array, new_pos_est, measured_time_shifts);

    // if new residual is smaller than previous residual, keep moving in that direction
    if (residual < prev_residual){
      pos_est = new_pos_est;
    }

    // otherwise, choose a new direction on next iteration
    else{
      residual = prev_residual
      new_dir_flag = 1;
    }
  }

  iter++;
}

// once an exit condition has been satisfied, break the loop
return pos_est;
}
```

Given a 3D position estimate of the transmitter, the 3D positions of each microphone, and the measured time shifts as inputs, the residual function returns the sum of squared differences between the measured time shifts and the expected (calculated) time shifts for that position estimate. Essentially, the function determines what the time shifts would be if the transmitter was at the 3D position given as input; then, it compares these expected time shifts to the measured time shifts. If the position estimate given is at the true position of the transmitter (and assuming no noise or error), then the residual function will return zero. The further that the position estimate is from the true position of the transmitter, the larger the residual is. For four microphones, the residual function is smooth and has few local minima; most are very close to the global minimum. This makes it easy for the HJS to converge on a minimum near the true transmitter location.

The residual function, as well its helper functions, can be seen below. The sum of squared differences is used (as opposed to the absolute difference) because it makes the residual function steeper; it also helps avoid any single time difference from being too far off.

The function `calculate_toa()` uses the 3D coordinates of each rotated microphone and the test position to calculate the Euclidean distance between each microphone and the test point. Then, it converts the distance to a time-of-arrival measurement using the speed of sound for the acoustic medium (in this case, air). The residual function subtracts each time-of-arrival from the first microphone's time-of-arrival to get the time differences. Lastly, it calculates the sum of squared differences between these estimated shifts and the measured time shifts using `calculate_squared_diff()`, and returns the value to the HJS function.

```c
float32_t calculate_residual(MicArray_t* mic_array, Point3D
    target_point, float32_t measured_time_shifts[]) {
  float32_t toa0 = 0;   /**< time of arrival for first mic */
  float32_t toaN = 0;   /**< time of arrival for Nth mic */
  float32_t* target_time_shifts = malloc((mic_array->num_mics - 1)
     * sizeof(float32_t));

  toa0 = calculate_toa(mic_array->base_points[0], target_point,
    sound_speed_mps);

  // for each mic, calculate the time of arrival (TOA)
  for (int i = 1; i < mic_array->num_mics; i++) {
    toaN = calculate_toa(mic_array->base_points[i], target_point,
    sound_speed_mps);

    // time shift is the first mic TOA minus the Nth mic TOA
    target_time_shifts[i-1] = toa0 - toaN;
  }

  // take the sum of squared differences between the time shifts
  float32_t residual = calculate_squared_diff(measured_time_shifts
    , target_time_shifts, mic_array->num_mics - 1);

  // save the calculated time shifts into a global array (for
    debugging / comparing to the real estimates)
  for (int i = 0; i < mic_array->num_mics - 1; i++) {
    calcd_time_shifts[i] = target_time_shifts[i];
  }

  // free the target time shift memory to avoid memory leaks
  free(target_time_shifts);
  return residual;
}

float32_t calculate_toa(Point3D pointA, Point3D pointB, float32_t
    sound_speed_mps) {
  float32_t xA = pointA.x;
  float32_t yA = pointA.y;
  float32_t zA = pointA.z;
  float32_t xB = pointB.x;
  float32_t yB = pointB.y;
  float32_t zB = pointB.z;
  float32_t euclidean_dist = sqrt((xA-xB)*(xA-xB) + (yA-yB)*(yA-yB
    ) + (zA-zB)*(zA-zB));
  float32_t toa = euclidean_dist / sound_speed_mps;
  return toa;
}

float32_t calculate_squared_diff(float32_t measured_data[],
    float32_t target_data[], int data_len){
  float32_t squared_diff = 0;
  for (int i = 0; i < data_len; i++){
    squared_diff += ((measured_data[i] - target_data[i])*(
    measured_data[i] - target_data[i]));
  }
  return squared_diff;
}
```

Lastly, the `arg_min()` function locates the index of the minimum value of the `residual_arr[]` list. This step is crucial for determining the best direction in the HJS algorithm.

```c
int arg_min(float32_t data_array[], int data_len){
  float32_t minValue = 1e10;
  int minIndex = 0;
  for (int i = 0; i < data_len; i++) {
    if (data_array[i] < minValue) {
      minValue = data_array[i];
      minIndex = i;
    }
  }
  return minIndex;
}
```

3.9.3 Incorporating Transmitter Position

Once an exit condition is met, the HJS function returns a position estimate to the main code. This estimate describes the position of the transmitter relative to the center of the receiver array. However, the transmitter position in the test frame is always known, using GPS coordinates in the full underwater implementation or measured 3D coordinates in this air-based implementation. These two measurements need to be combined.

Since the position estimate from the HJS algorithm gives the relative position of the transmitter from the iSBL array's center, the inverse of this vector describes the location of the receiver array relative to the transmitter. This inverse can be added to the transmitter's true position to get the estimated position of the iSBL array in the test frame. For a one-dimensional example: if a receiver determines a transmitter is located 3 meters ahead of itself in the x-direction, and it knows that the transmitter is located at `x=5m`, then the receiver must be located at `x=2m`. The code below implements this on the STM32 using the 3D position estimates.

```c
float true_x = 3.880f;
float true_y = 0.0f;
float true_z = -0.682f;
...
```

```
// perform hooke-jeeves search to minimize the residual function
raw_isbl_pos_est = hooke_jeeves_search_2d(rotated_mic_array,
   init_pos_est, measured_time_shifts, 1000, 1e-13, 1e-6, 2.0f);
...
// subtract estimated position to transmitter from known
   transmitter location to get receiver position
raw_isbl_pos_est.x = true_x - raw_isbl_pos_est.x;
raw_isbl_pos_est.y = true_y - raw_isbl_pos_est.y;
raw_isbl_pos_est.z = true_z - raw_isbl_pos_est.z;
```

Finally, the acoustic position estimate for a single iteration is saved in the variable `raw_isbl_pos_est[]`. From recording a single acoustic pulse on the four receivers, measuring the time shift between microphones, rotating the microphones' 3D positions using the current orientation, running 2D HJS with the simulated depth measurement, and reflecting the position estimate across the transmitter, a single acoustic position estimate is formed! In Chapter ??, this position estimate is improved using Kalman filtering.

Bibliography

[1] "Underwater acoustic positioning system," Academy of Positioning Marine And Bathymetry, Tech. Rep., 2007. [Online]. Available: https://www.hydro-international.com/files/3fb82f3961d1f9890fc2a475cf56d9ed.pdf

[2] K. Vickery, "Acoustic positioning systems "a practical overview of current systems"." Dynamic Positioning Conference, 1998. [Online]. Available: https://dynamic-positioning.com/proceedings/dp1998/SVickery.PDF

[3] Y. Wu, X. Ta, R. Xiao, Y. Wei, D. An, and D. Li, "Survey of underwater robot positioning navigation," *Applied Ocean Research*, vol. 90, p. 101845, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141118718305546

[4] S. Liu, H. Guo, Z. Qian, J. Li, X. Wang, W. Sun, L. Zhang, and A. Zhang, "A study on a novel inverted ultra-short baseline positioning system and phase difference estimation," *Journal of Marine Science and Engineering*, vol. 11, no. 5, 2023. [Online]. Available: https://www.mdpi.com/2077-1312/11/5/952

[5] Y. Wang, R. Hu, S. H. Huang, Z. Wang, P. Du, W. Yang, and Y. Chen, "Passive inverted ultra-short baseline positioning for a disc-shaped autonomous underwater vehicle: Design and field experiments," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 6942–6949, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9783050

[6] J. Luo and H.-L. Ko, "Ukf-based inverted ultra-short baseline slam with current compensation," *IEEE Access*, vol. 10, pp. 67 329–67 337, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9804706

[7] K. Key and S. Constable, "Inverted long-baseline acoustic navigation of deep-towed csem transmitters and receivers," *Mar Geophys Res*, vol. 42, no. 6, feb 2021. [Online]. Available: https://link.springer.com/article/10.1007/s11001-021-09427-z

[8] A. Alcocer, P. Oliveira, and A. Pascoal, "Underwater acoustic positioning systems based on buoys with gps." Eighth European Conference on Underwater Acoustics, 8th ECUA, jun 2006. [Online]. Available: https://welcome.isr.tecnico.ulisboa.pt/wp-content/uploads/2015/05/1502_ECUA06_LEGAL.pdf

[9] S. M. Smith and D. Kronen, "Experimental results of an inexpensive short baseline acoustic positioning system for auv navigation,"

1997. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=634454

[10] Z. Liu, T. Yang, W. Xu, J. Yu, D. M. McFarland, and H. Lu, "Underwater acoustic positioning with a single beacon and a varied baseline for a multi-jointed auv in the deep ocean," *IET Radar, Sonar, and Navigation*, vol. 14, no. 5, pp. 669–676. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-rsn.2019.0330

[11] M. Morgado, P. Oliveira, and C. Silvestre, "Tightly coupled ultrashort baseline and inertial navigation system for underwater vehicles: An experimental validation," *Journal of Field Robotics*, vol. 30, no. 1, pp. 142–170. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21442

[12] A. Dikarev, D. Hegner, and Z. Ivan, "Ucnlnav," 2021. [Online]. Available: https://github.com/ucnl/UCNLNav

[13] STMicroelectronics, *STM32H723/733, STM32H725/735 and STM32H730 Value line advanced Arm®-based 32-bit MCUs, Reference Manual*, 2021. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0468-stm32h723733-stm32h725735-and-stm32h730-value-line-advanced-armbased-32bit-pdf

[14] ——, "https://www.st.com/en/evaluation-tools/nucleo-h723zg.html," 2024. [Online]. Available: https://www.st.com/en/evaluation-tools/nucleo-h723zg.html

[15] F. Jensen, W. Kuperman, M. Porter, and H. Schmidt, *Computational Ocean Acoustics*. American Institute of Physics, 1994.

[16] ElectronicsTutorials, "Active band pass filter," 2024. [Online]. Available: https://www.electronics-tutorials.ws/filter/filter_7.html

[17] Bourns, "Standard values used in capacitors, inductors and resistors," 2024. [Online]. Available: https://www.bourns.com/support/technical-articles/standard-values-used-in-capacitors-inductors-and-resistors

[18] P.-W. E. Corporation, "Air ultrasonic ceramic transducers 400st/r160," 2024. [Online]. Available: https://www.mouser.com/datasheet/2/911/T400S16-1369697.pdf

[19] T. Gravekamp, "Weact black pill v2.0," 2024. [Online]. Available: https://stm32-base.org/boards/STM32F411CEU6-WeAct-Black-Pill-V2.0.html

[20] J. Frabosilio, "Deciphering ultrasonic audio transmissions using neural networks," jun 2023.

[21] S. Dixon, L. Kang, A. Feeney, and W. E. Somerset, "Active damping of ultrasonic receiving sensors through engineered pressure waves," *Journal of Physics D: Applied Physics*, vol. 54, no. 13, p. 13LT01, jan 2021. [Online]. Available: https://iopscience.iop.org/article/10.1088/1361-6463/abd582

[22] S. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing.* California Technical Publishing, 1997.

[23] Cmglee, "Visual comparison of convolution, cross-correlation and auto-correlation," nov 2016. [Online]. Available: https://en.wikipedia.org/wiki/Cross-correlation#/media/File:Comparison_convolution_correlation.svg

[24] J. Krech, "Cmsis dsp library," 2015. [Online]. Available: https://github.com/ARM-software/CMSIS_4/blob/master/CMSIS/Include/arm_math.h

[25] D. Rose, "Rotation quaternions, and how to use them," may 2015. [Online]. Available: https://danceswithcode.net/engineeringnotes/quaternions/quaternions.html

[26] J. Roberts, M. Kochenderfer, and R. Trochet, "Minimizing in 3d, method 3: Hooke-jeeves," 2014. [Online]. Available: https://web.stanford.edu/group/sisl/k12/optimization/MO-unit2-pdfs/2.11minimum3D2hooke-jeeves.pdf

[27] R. Hooke and T. A. Jeeves, "Direct search solution of numerical and statistical problems," *Journal of the ACM*, vol. 8, apr 1961. [Online]. Available: https://dl.acm.org/doi/10.1145/321062.321069

[28] V. Torczon, "Pattern search methods for nonlinear optimization," *SLAG/OPT Views-and-News*, 1995. [Online]. Available: https://www.cs.wm.edu/~va/articles/viewsNnews.pdf

[29] B. Robotics, "Bar30 high-resolution 300m depth/pressure sensor," 2024. [Online]. Available: https://bluerobotics.com/store/sensors-cameras/sensors/bar30-sensor-r1/