CPX 3 Report

ECE 434: Digital Signal Processing

November 23, 2024

C1C Victor Chen

C1C Ben Cometto

C1C Nick Csicsila

C1C Geoff Stentiford

Contents

List of Figures

List of Tables

Abstract

Sonar, sending and receiving sounds as a sensor, is a valuable tool that electrical and computer engineering provides. In this project, CPX 3, we are improving upon a provided system. The system collected data from a phased array of four omni-directional microphones, after a pulse from a single approximately omni-directional speaker. We are working to improve both the speed and quality of the processing of this data, using the digital signal processing and problem solving techniques learned this semester. [REWRITE THIS ONCE REST IS WRITTEN AND WRITE MORE ABOUT METHODS AND RESULTS]

1 Introduction

Sonar, ultrasound, and radar are all based on the same principle: transmit a wave, and record its echos. The only differences are the type of wave or the frequencies involved. In this project, we are using sonar. Working in the audible range enables the use of low-cost, common speakers and microphones. Additionally, it allows us to hear to pulses and echos, which can serve as another problem solving tool.

For this project, we are working with recorded data. The recorded data was produced using a single omni-directional speaker to transmit the pulse and a phased array of four omni-directional microphones to receive the echos. Each microphone is given a channelL X_1, X_2, X_3, and X_4.

The signal processing is done inside of a MATLAB script, following the process diagrammed in Figure 1. **NEEDS TO BE REDRAWN WITH OUR NEW ORDER**

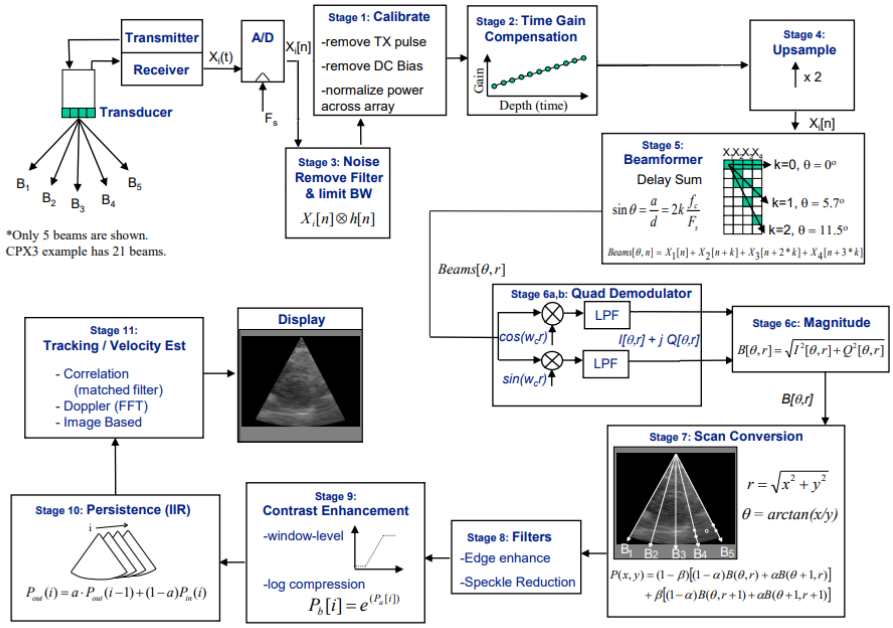

Figure 1: Signal Processing Stages

Each stage is described in more detail, to include its purpose, design goals, implementation, and analysis results, in its respective section.

2 Block 1: Noise Removal, Bandwidth Limiting, and Bias Correction

This block carries out three functions: it removes the DC bias inherent to the sensors, reduces the noise in the signal, and limits the bandwidth to only that which is needed. This is accompished in three stages: preliminary debiasing through the very simple method of subtracting the average value of each sensor channel, a high-pass filter to fully remove bias and very low-frequency noise, and a low-pass filter to clamp the bandwidth at 20kHz and remove any high-frequency noise.

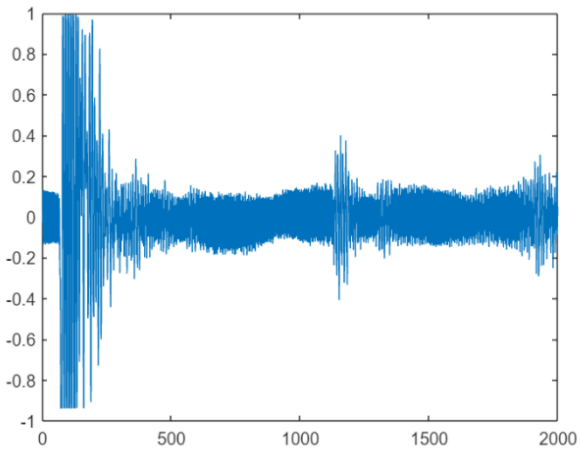

Figure 2: Pre-filtered average subtracted only

The low-pass filter was a 30-order Tukey window filter with an alpha of 0.5 and a cutoff frequency of 22.16kHz. Its stopband lobes are aligned such that the 25kHz "jamming" falls directly into one of the cracks. The high-pass filter, meanwhile, is a 61-order least-squares filter with a cutoff which targets only the DC component. Using two cascaded filters resulted in a lower total order than a single bandpass filter.

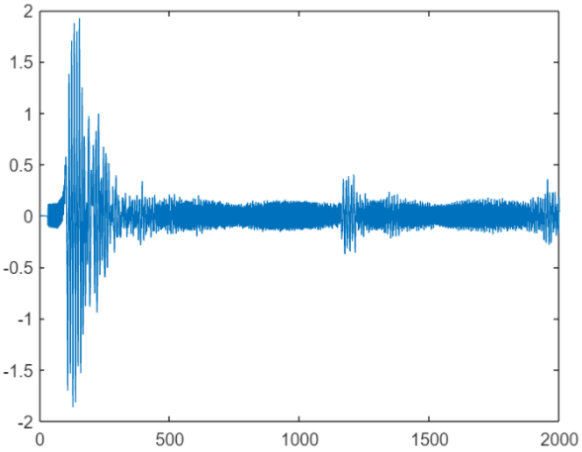

Figure 3: High-pass filtered, DC bias removed

An infinite impulse response (IIR) filter, specifically an elliptic filter, would allow for a much lower-order

filter which would be faster to run, but the need to preserve phase relations meant we restricted ourselves to finite impulse response filters to be on the safe side, as FIR filters ensure a linear phase response. Elliptic IIR filters do not distort phase too badly, but enough that some amount of distortion is visible in the final display.

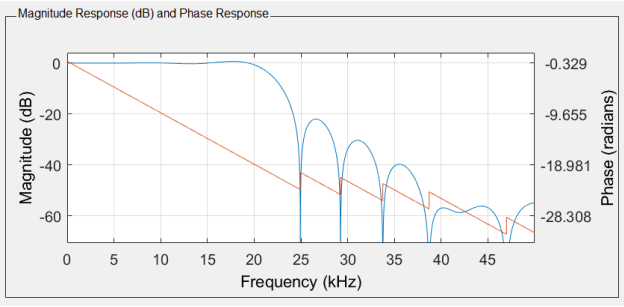

Figure 4: Pre-filtered average subtracted only

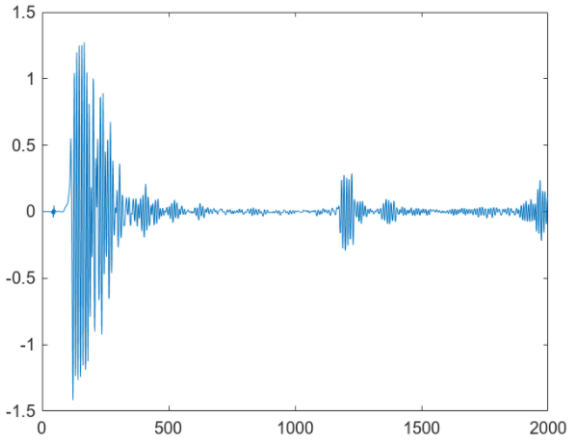

Figure 5: Low-pass filtered, high frequencies removed

We decided to place this block first, as the noise-reduced signal is much simpler to deal with when implementing the calibration stage than the raw data signal.

3 Block 2: Calibration

The calibration block detects the sonar transmissions, shifts the data in the time domain so that the start of the arrays line up with the transmit time, zeros out the transmission, and normalizes the received signal across the arrays. To determine where the transmission occurs, the block looks for 10 consecutive samples where the absolute value of the signal exceeds 0.08. Then, to determine where it ends, it waits for 30 consective samples with a level below 0.04. From there, it shifts the data left to align the array start with the transmission's start.

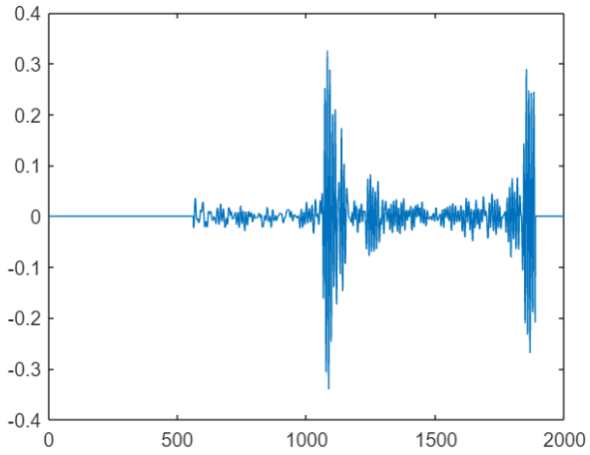

Figure 6: Transmission zeroed

To normalize the signal magnitudes, the block examines four signals to find the maximum absolute values in each and applies the necessary gain to bring each signal up to the same strength as The strongest signal.

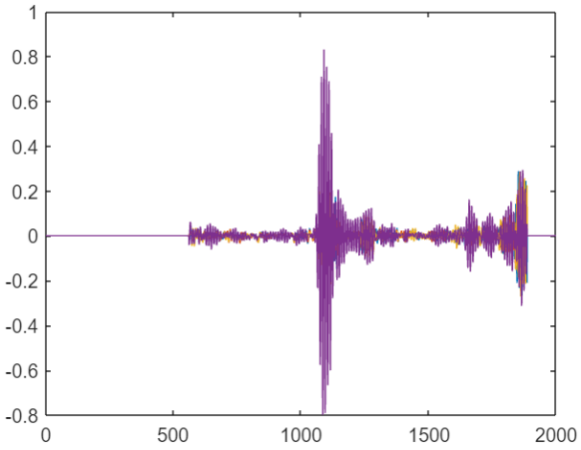

Figure 7: All channels, pre-normalization

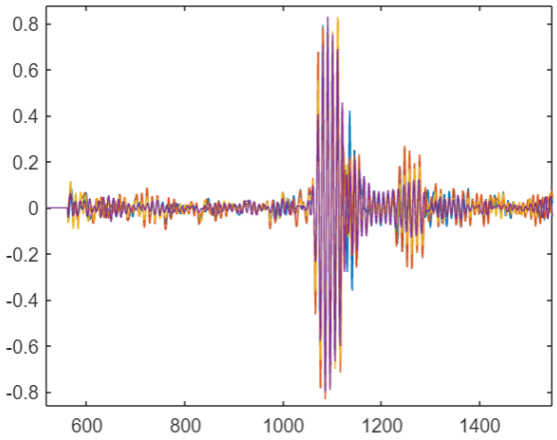

Figure 8: Channels normalized

To optimize for speed, the calibration stage was made into a unified module instead of a distinct stage 1a and stage 1b. By using the already-filtered signal, a much simpler (and therefore faster and less error-prone) approach was feasible.

4 Block 2 : Time Gain Compensation

4.1 Theory

As sound travels, it attenuates through geometric spreading. As a result, the magnitude of the second reflected pulse is significantly lower. Therefore, the program must compensate to increase the magnitudes of the two pulses back to the original magnitude of 1. The samples must first be converted into distance. This is easily accomplished using the following conversion.

$$r = \frac{\text{Sample Index}}{F_s} \cdot c_{\text{sound}} \tag{1}$$

For omni directional transducers the inverse-square law ($I \propto \frac{1}{r^2}$) dictates how the signal degraded with time. However, this signal was relatively directional and only degraded by r. I order to reduce the effects, the samples were multiplied by r. Thus using a linear function to increase the magnitude of the first and second pulses.

4.2 Analysis

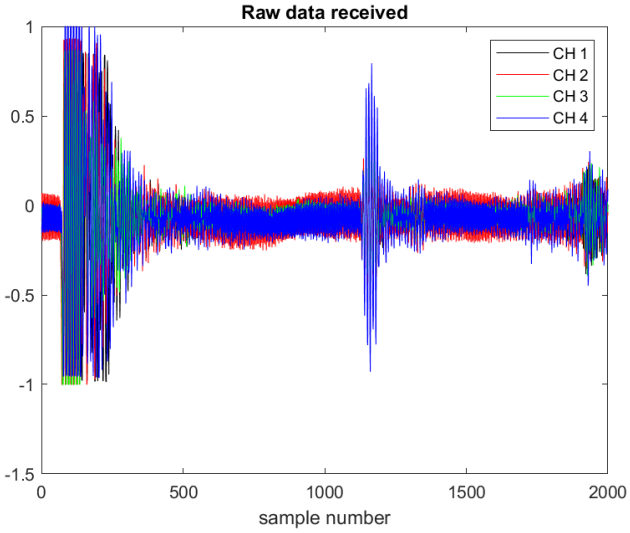

Figure 9: Raw Data

Figure 9 shows the initial signal detected. The goal of Block 2 is boost the first and second pulses to a higher magnitude to increase clarity. As mentioned before, the pulses were boosted by multiplying the whole signal by distance traveled r.

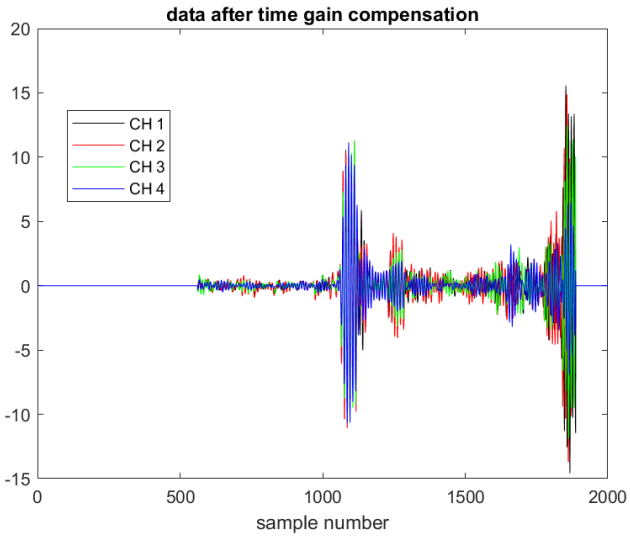

Figure 10: Data after time gain compensation

Figure 10 shows the transformation after the attenuation. As seen, the first and second pulses are much closer in relative magnitude. Before, pulse 2 would be hidden due to its low magnitude.

4.3 Performance

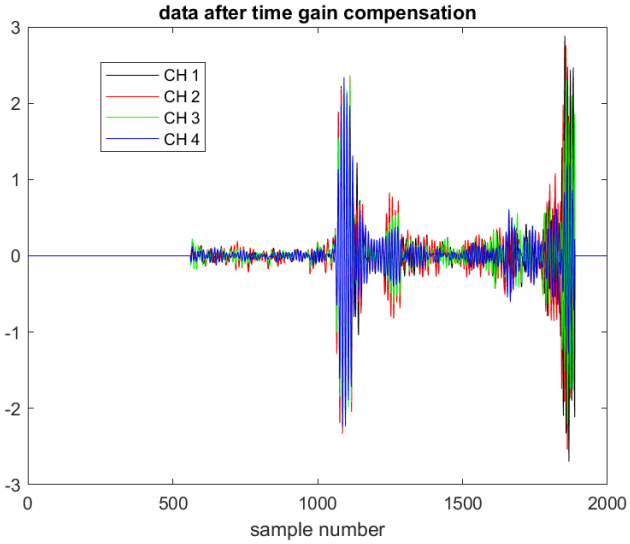

Figure 11: Data after time gain compensation provided

Function	Time (μs)
Provided `.p`	1435
Rewritten `.m`	56.5

Table 1: Performance of TGC

As seen in Table 1, this implementation performed much quicker than the provided example. In addition it provided higher quality for the second pulse as seen in Figure 11. The implementation was very simple, aiding its performance. The approach was creating an array for each channel and filling it with the values of the samples. Those values were then multiplied by r. The function timeGainCompensation then only had to multiply the data by the new array. This increased the speed by 184%.

5 Block 4: Upsampling (2x)

Upsampling involves increasing the number of samples by interpolating the existing data. In this case, we are upsampling our data so that we are able to form more beams, and thus have a higher resolution image.

The goal for the upsampling block is to double the number of samples. In order to do this, there are two steps. First, zeros must be added in between the datapoints. Second, the zero-padded data must be low pass filtered with

$$f_c = \frac{F_{s,old}}{2} = 50\,\text{kHz}$$

to remove the reflected image of the data.

5.1 Implementation

We must implement both of these steps in a very fast way, without sacrificing quality.

First, MATLAB is meant for working with vectorized functions, and has vectorized methods of replacing certain indexes. Thus, the quickest way to add a zero after every sample (double the number of samples) is to replace every other entry of a matrix of zeros with the data point.

Second, in order to interpolate, we will use the minimum order filter possible that preserves the necessary information, because a lower order results in less multiplies and thus a quicker function. Additionally, FIR was chosen in order to preserve the relative phase of the received signals, which is necessary for preventing distortion in the sonar image. Equirriple was used as the design method.

In this case, an order 3 filter with the specifications in Table 2 works. Our information is contained in the band less than 20 kHz , and thus we can set f_{pass} to 20 kHz . Then, because our $F_{s,old} = 100\,\text{kHz}$, there is a significant amount of bandwidth between our data's frequency spectrum and its reflected image. Thus, we can set f_{stop} to be 69 kHz , the sharpest it can be while still being order 3. Through various trials, an A_{stop} of 40 dB sufficiently suppresses the image, will also allowing the filter to be order 3.

Order	Fpass	Fstop	Apass	Astop
3	20 kHz	69 kHz	1 dB	40 dB

Table 2: Parameters for Upsampling Low Pass Filter

The frequency response of the low pass filter can be seen in Figure 12.

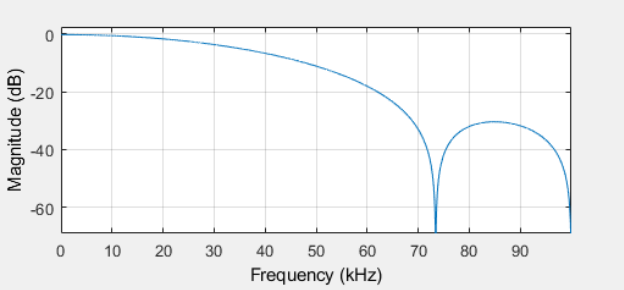

Figure 12: Frequency Response of Upsampling Low Pass Filter

In order to preserve this speed, the function header was modified to accept the numerator taps of the low pass filter, matrix of zeros, and the length of the matrix of zeros as input in addition to the data to be upsampled.

5.2 Analysis

An initial, simple test to confirm that the function upsamples properly is by inspecting a linear function. This can be seen in Figure 13.

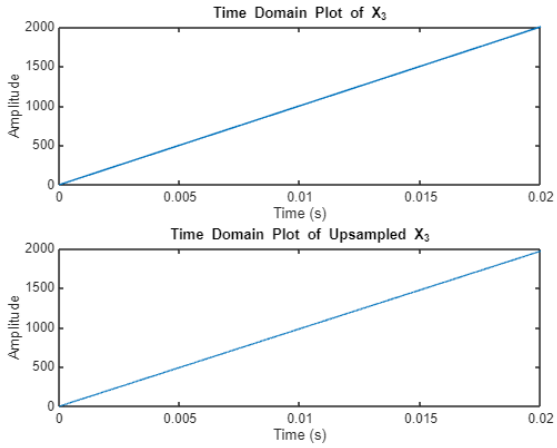

Figure 13: Upsampled Linear Function

It can be seen in Figure 14 that the upsampled version contains the same data, but has twice as many points.

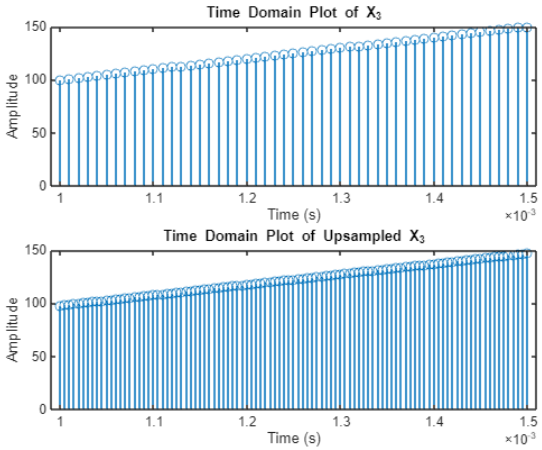

Figure 14: Zoomed Portion of Upsampled Linear Function

Testing on various other types of synthetic data, such as sinusoids and linear combinations of sinusoids, further confirmed that the function was implemented as intended.

As the next test, we can upsample the provided test data (unmodified by the other stages) to ensure it has no unintended effects. As can be seen in Figure 15 and Figure 16, te data was interoplated correctly. Each channel is represented by a color.

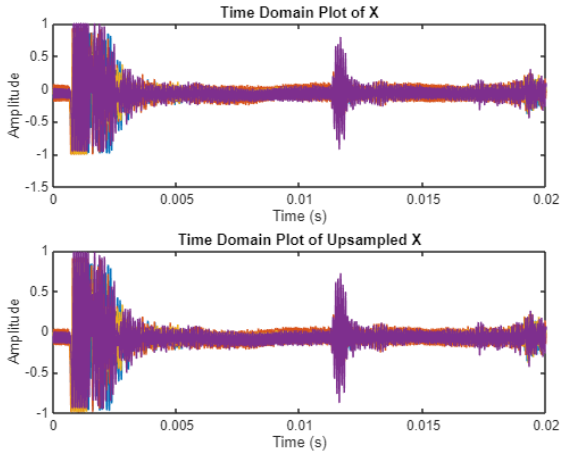

Figure 15: Upsampled Data in Time

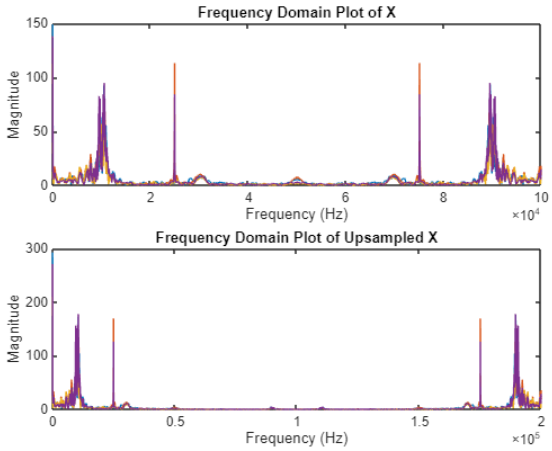

Figure 16: Upsampled Data in Frequency

Zooming in for Figure 17 and Figure 18, we can more clearly see that the interpolation was successful.

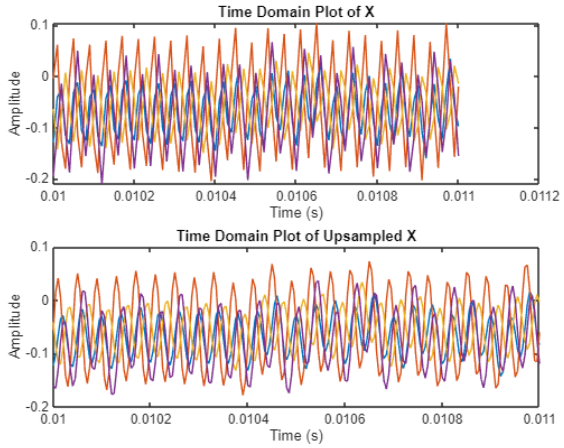

Figure 17: Zoomed Portion of Upsampled Data in Time

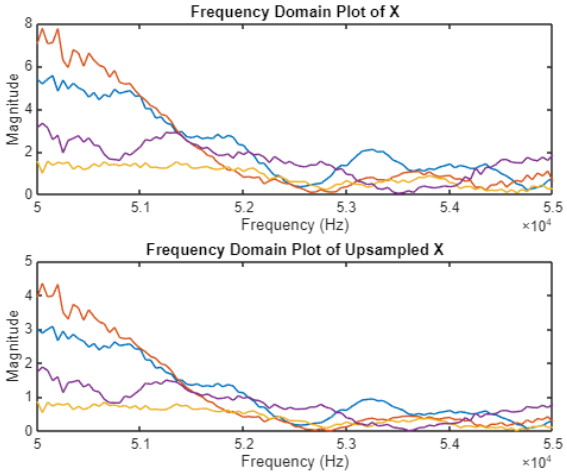

Figure 18: Zoomed Portion of Upsampled Data in Frequency

Therefore, the function successfully interpolates the data at a high quality. The next step in the analysis is in the timing. Benchmarking is difficult, but here we are only comparing the newly written function to the provided `.p` file. In this case, each function was timed (using `tic` and `toc`) over $10,000$ iterations, and the average runtimes can be seen in Table 3. The same data (the provided test data) was used, and the computer was in as similar a state as possible.

Function	Time (μs)
Provided `.p`	5455
Rewritten `.m`	64.7

Table 3: Parameters for Upsampling Low Pass Filter

Therefore, the rewritten `upsampling.m` function provides a high quality upsampling by 2 and is about 100 times faster than the original function.

6 Block 5: Beamforming

Beamforming is a signal processing technique used to spatially direct signals, enhancing desired signals while suppressing other signals with interference. A delay-and-sum beamforming algorithm is applied to the four-channel system of uniformly spaced linear sensor array of four omnidirectional microphones. Assuming the signal source is in the far field, the incoming wave fronts are approximately linear, enabling the computation of beam angle for each integer sample delay k.

The goal for the beamforming block is to use the delay-and-sum beamforming function

$$Beams[k, n] = X_1[n] + X_2[n + k] + X_3[n + 2 * k] + X + 4[n + 3 * k]$$

to spatially focus and enhance signals from a specific direction across our four-channel system.

6.1 Implementation

This block is designed to perform beamforming across 4,000 samples from the four channels. Given the need for multiple iterations, it is crucial to balance between processing speed and output quality.

To optimize the beamforming function, we utilize MATLAB's linear vectorization and precomputing capabilities, enabling faster and more efficient processing by operating on entire data arrays simultaneously instead of relying solely on iterative loops.

```
% Create array to hold beam vals
beams = zeros(21, FrameSize);   % Initialize beamformed array

% Precompute offsets for all sensors and k values
k_offsets = [k_range', 2 * k_range', 3 * k_range']; % Precompute offsets
```

Figure 19: Precomuting Equations

The first optimization method involves precomputing the array to store the beams and the offsets for all sensors. This approach preloads the necessary matrices, eliminating the need for dynamic resizing or appending during runtime, which can significantly slow down the program. As illustrated in Figure 1, the k_offsets for the delays are computed in a single step using vectorized operations. Additionally, a beams array is preallocated in advance to hold the calculated beams, ensuring optimal memory usage and faster processing.

```
% Apply beamforming equation (vectorized for all valid n)
beams(kk, n_range) = data2(n_range, 1) + ...
                     data2(n_range + k_offsets(k,1), 2) + ...
                     data2(n_range + k_offsets(k,2), 3) + ...
                     data2(n_range + k_offsets(k,3), 4);
```

Figure 20: Beamforming Computation

The second optimization method involves performing beamforming calculations across all 4,000 samples simultaneously, rather than iterating through each index individually. This reduces the number of loops, making the process much more efficient.

6.2 Analysis

After developing the beamforming function, test data was generated to evaluate its performance. The test data consisted of four channels of in-phase sine waves, defined by the equation $sample = \sin(t/40)$, where t ranged from 1 to 4,000. After ensuring the function works with the test data, I used data2 directly from the project.

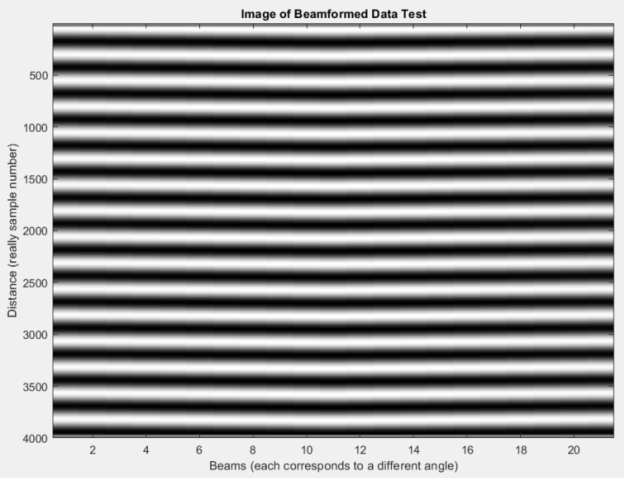

Figure 21: Output for Testing with sin(t/40) wave

As shown in Figure 3, each black-and-white horizontal line represents a wave. With 4,000 samples in the equation $\sin(t/40)$, this corresponds to approximately 16 waves. The figure confirms this, demonstrating that the beamforming function performs as expected. It is important to note that the waves are evenly horizontal due to the in-phase channels, however, as the frequency of the waves increases, the evenness of the waves will begin to separate.

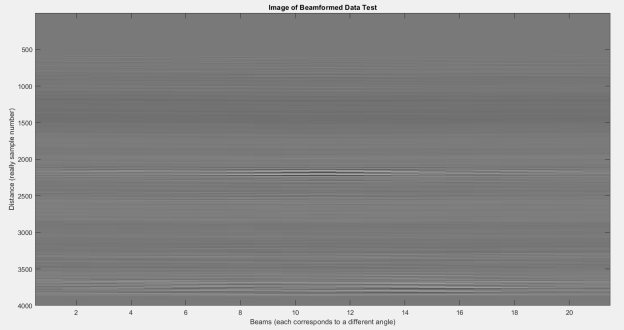

Figure 22: Beamform Output from Data

As shown in Figure 4, the beamform output of data2 reveals two regions where the signal strength is highest. These are represented by two prominent white "blobs"–one near the center and another near the bottom-right corner of the beamformed image. These regions indicate areas where the waves return well-constructed, suggesting the presence of an object that the waves interacted with.

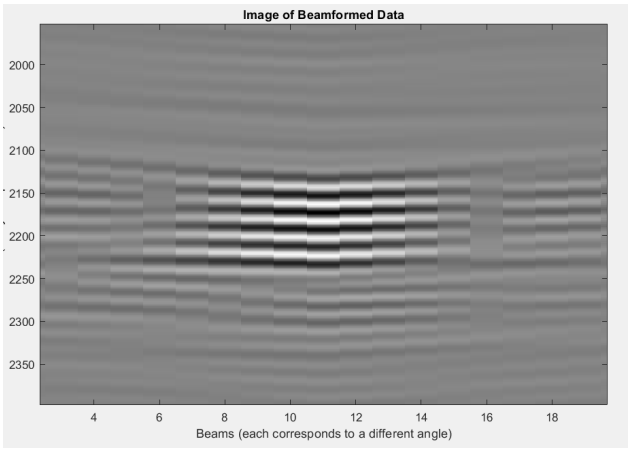

Figure 23: Zoomed In Beamform Output

By zooming into the center of the beamform image in Figure 5, we can see that while the signal strength is high, each beam is still slightly offset. This offset may be due to inaccuracies in my delay calculations or it might be due to the geometry/spacing of the sensors. Plenty of time was spent on trying to correct the offset by shifting beams, however none were successful. More work will need to be done to correct this issue.

The beamforming function achieved an average execution time of 0.0023155 seconds, a substantial improvement compared to pre-optimization runs, which took tens of seconds. This demonstrates a significant increase in computational efficiency for beamforming.

7 Conclusion

[Needs to be written]

Documentation

We worked as a team on this project. Additionally, Ben Cometto used resources such as the mathworks website for MATLAB syntax help (ex, confirming how to index every other entry of a matrix) and had a discussion with ChatGPT regarding recording phase, available here.