CPX 3 Report

ECE 434: Digital Signal Processing

December 7, 2024

C1C Victor Chen

C1C Ben Cometto

C1C Nick Csicsila

C1C Geoff Stentiford

Contents

List of Figures

List of Tables

Abstract

Sonar, using the data recorded by a phased array of four omni-directional microphones following an omni-directional pulse, allows objects to be detected and imaged. By calculating the values along a beam (direction), we are able to determine where the object that reflected the sound is located in relation to the sensors. However, in order to do this, several pre-proessing steps are required. First, we condition the data: reduce the noise, remove the DC bias, and limit the bandwidth. Next, we calibrate the data by lining up the recording with the transmission of the pulse and ensuring each channel is recorded evenly. Next, we compensate for the time-gain by reversing the inverse-square law. Then, we upsampled by 2. Finally, we form the beams. This preprocessing forms the backbone of the sonar system, and is the foundation for the future processing of the beam data into a useable image.

1 Introduction

This report presents the design process and outcomes for the development of a sonar system by Three Engineers and Ben Corporation. Leveraging advanced digital signal processing (DSP) techniques and MATLAB design tools, our team implemented an 11-stage sonar processing system. Throughout the project, we faced critical engineering decisions to balance the trade-offs between optimizing the system for high-quality sonar imaging and ensuring fast, real-time display capabilities.

The following sections provide a detailed account of our design approach, implementation, and analysis for each stage of the system. By documenting our decisions and their rationale, this report highlights the innovative solutions and technical expertise that distinguish our sonar processing system.

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 the 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 channel, labeled 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.

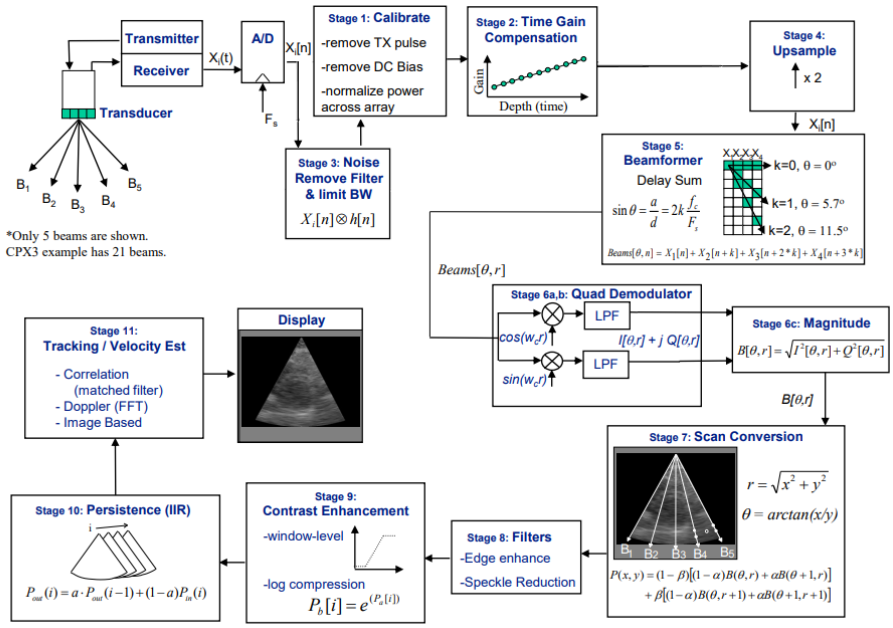

Figure 1: Signal Processing Stages

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

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.

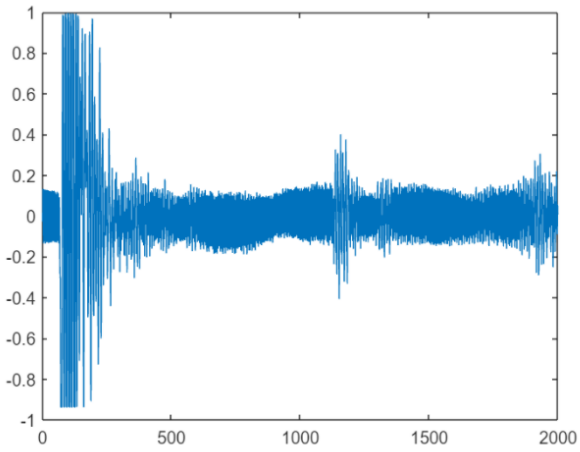

Figure 2: Pre-filtered average subtracted only

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.

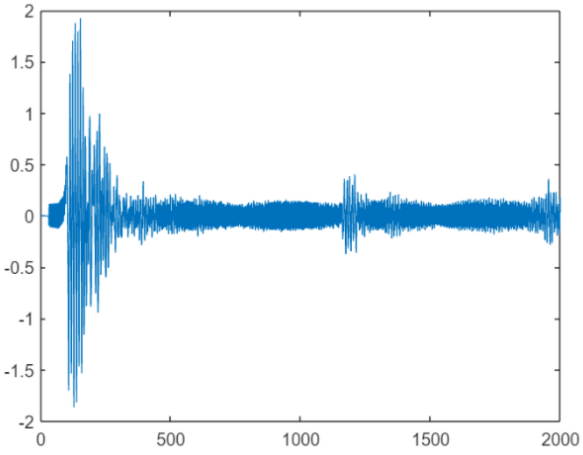

Figure 3: High-pass filtered, DC bias removed

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.

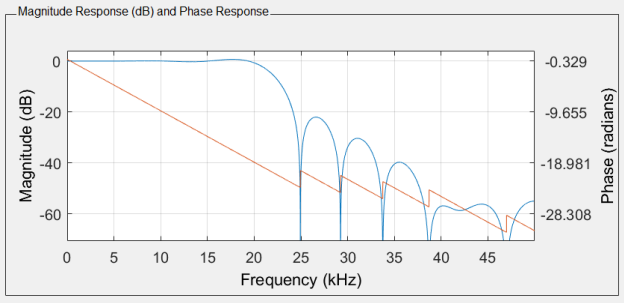

Figure 4: Pre-filtered average subtracted only

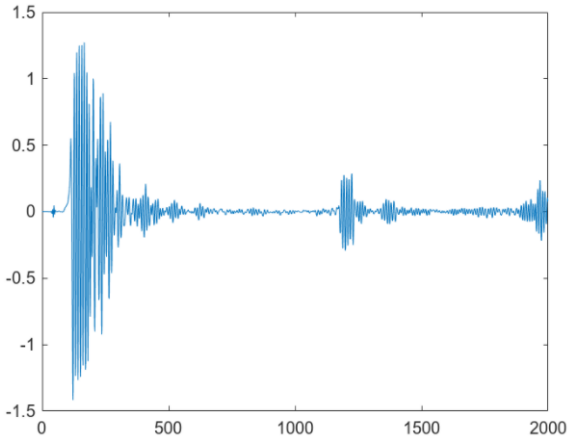

Figure 5: Low-pass filtered, high frequencies removed

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 - Stentiford

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.

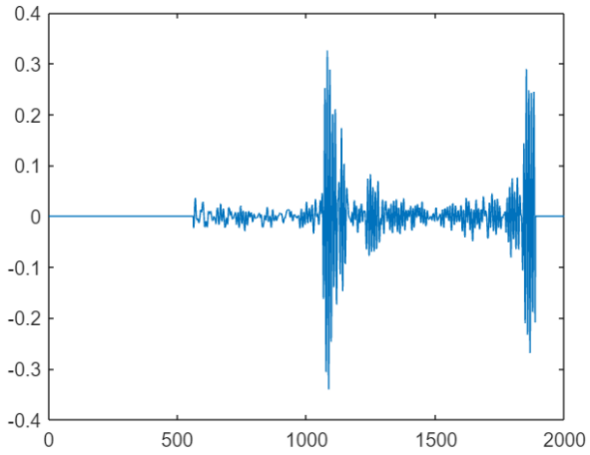

Figure 6: Transmission zeroed

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.

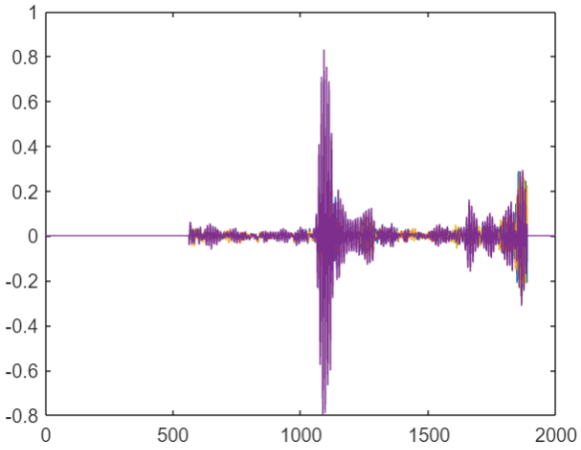

Figure 7: All channels, pre-normalization

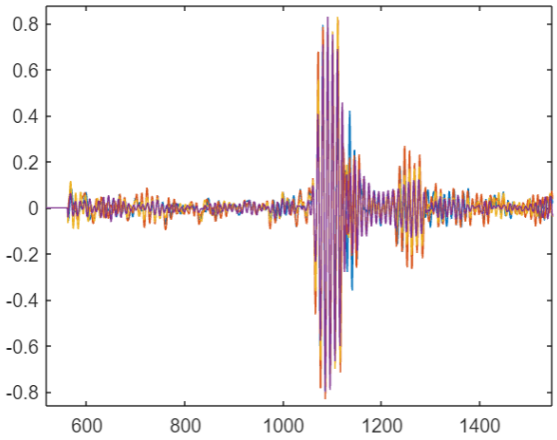

Figure 8: Channels normalized

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 - Csicsila

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. In order to reduce the effects, the samples were multiplied by $1 + r$. Thus using a linear function to increase the magnitude of the first and second pulses.

4.2 Analysis

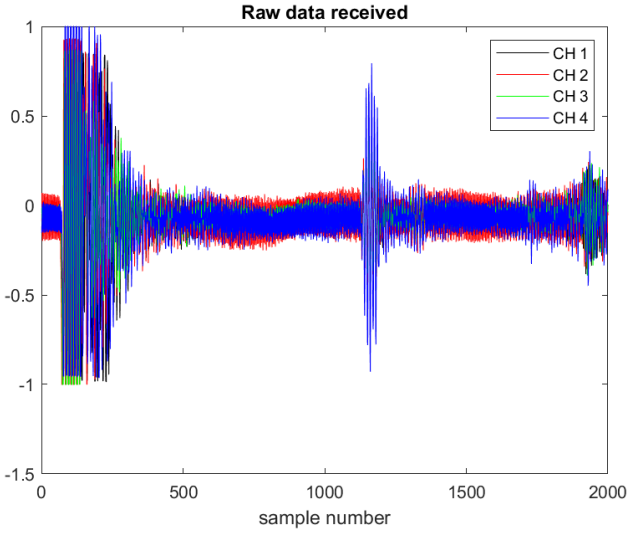

Figure 9: Raw Data

Figure 9 shows the initial signal detected. The goal of Block 2 is to 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.

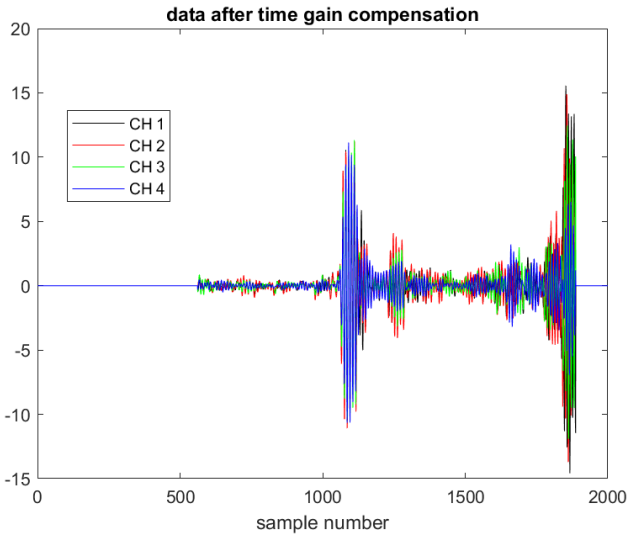

Figure 10: Data after time gain compensation

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

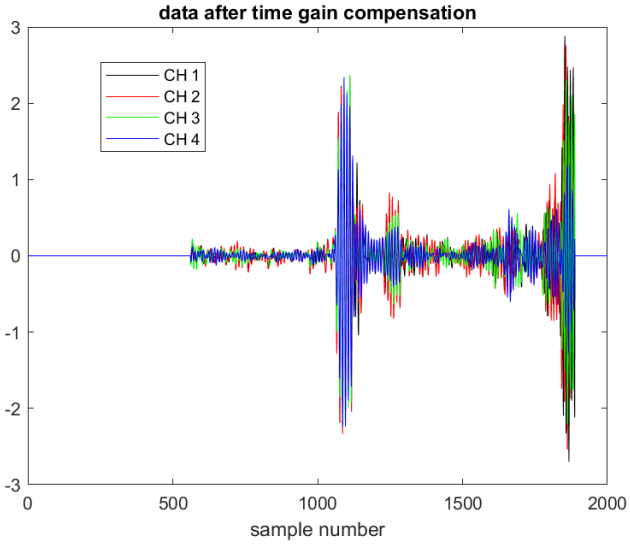

Figure 11: Data after time gain compensation provided

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) - Cometto

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.

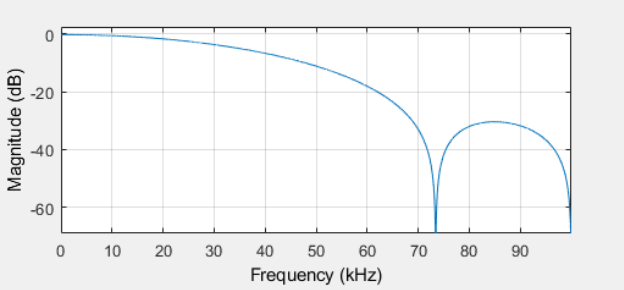

Figure 12: Frequency Response of Upsampling Low Pass Filter

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.

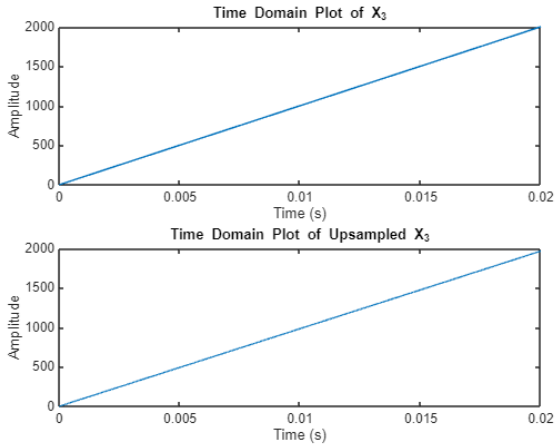

Figure 13: Upsampled Linear Function

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

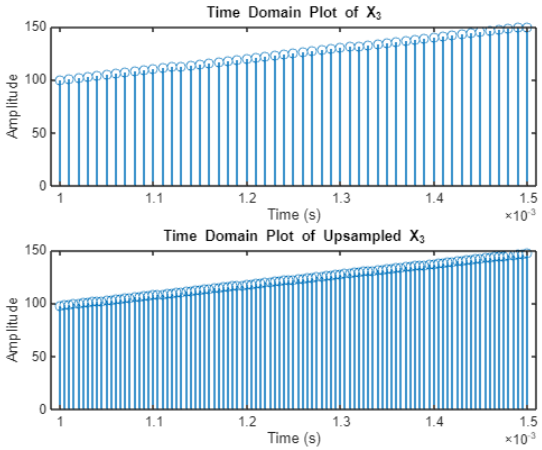

Figure 14: Zoomed Portion of Upsampled Linear Function

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, the data was interoplated correctly. Each channel is represented by a color.

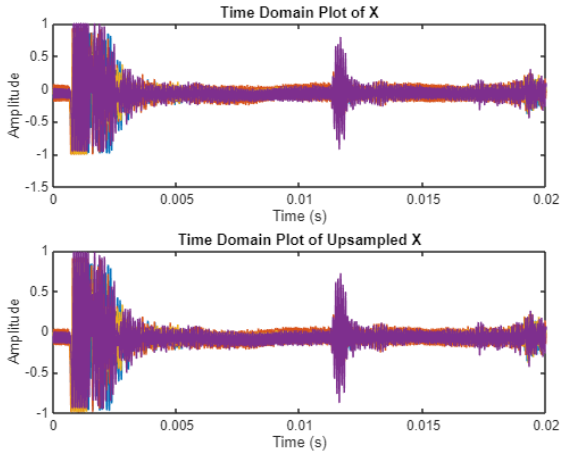

Figure 15: Upsampled Data in Time

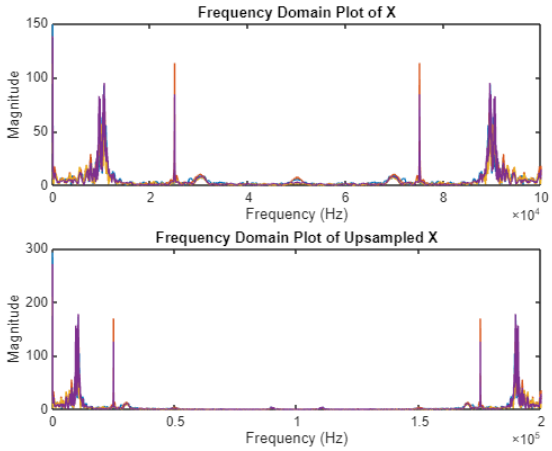

Figure 16: Upsampled Data in Frequency

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

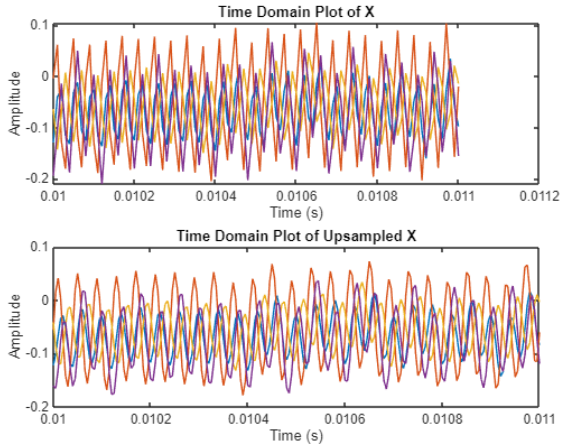

Figure 17: Zoomed Portion of Upsampled Data in Time

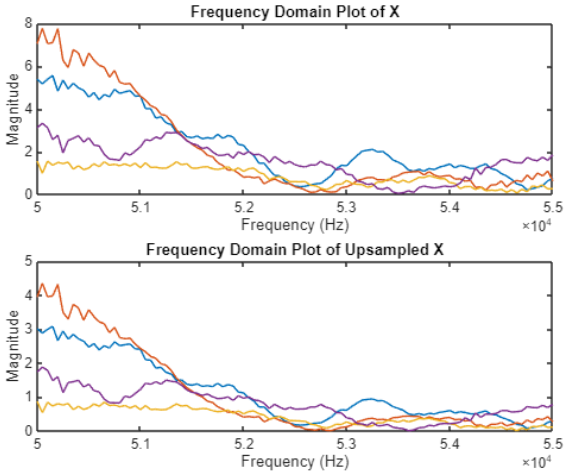

Figure 18: Zoomed Portion of Upsampled Data in Frequency

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: Provided vs Rewritten Upsampling Function Performance

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 - Chen

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.

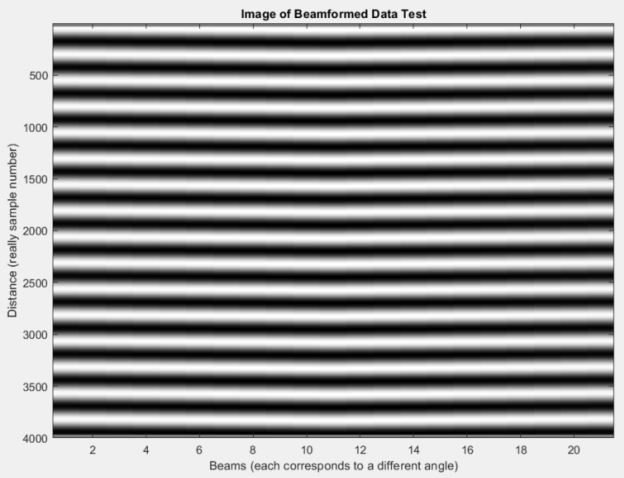

Figure 21: Output for Testing with sin(t/40) wave

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.

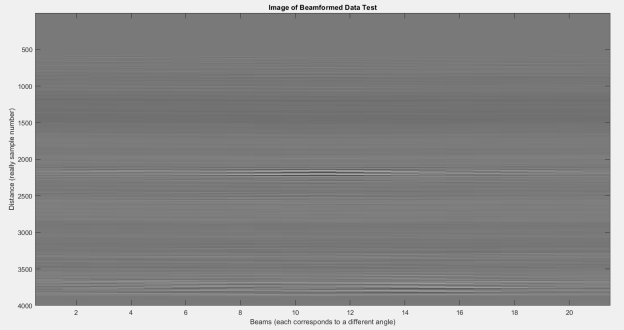

Figure 22: Beamform Output from Data

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.

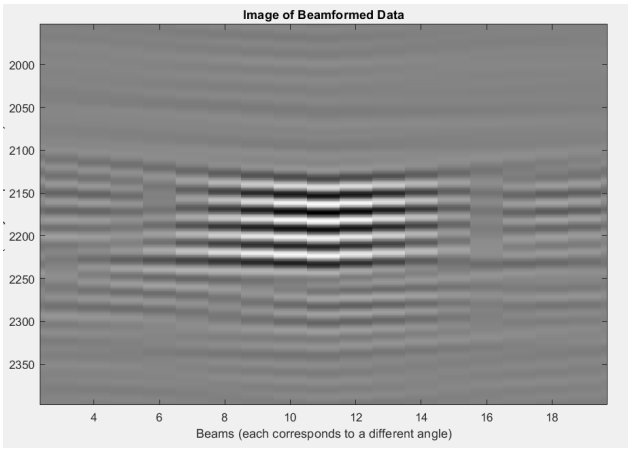

Figure 23: Zoomed In Beamform Output

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 Block 6: Quadrature Demodulation - Chen & Stentiford

The quadrature demodulator is used to detect the pulse envelopes. It is a type of amplitude modulation (AM) decoder which is able to function without knowing the exact phase of the signal's carrier wave. It consists of three stages: mixing, filtering, and magnitude calculation.

The mixing stage of a normal (synchronous) demodulator multiplies the received signal by the original carrier wave, matching the frequency and phase. Because we do not know the phase in this case, we instead split the incoming signal, multipling each copy by carrier waves 180° apart. Or, in mathematical terms:

$$y[n] = x[n] * (\cos(2\pi fn) + j\cos(2\pi fn + \pi)) = x[n] * (\cos(2\pi fn) + j\sin(2\pi fn))$$

To mitigate the relative slowness of trigonometric calculations, we pre-computed a pair of tables for cosine and sine which could be used in the actual block instead of calls to MATLAB's cosine and sine functions. The tables additionally already account for the sampling rate and carrier frequency in order to minimize multiplication operations inside the time- sensitive block.

After filtering, to merge the imaginary and real components back into a singal envelope, we simply take the magnitude:

$$Y = \sqrt{I^2 + R^2}$$

This is achieved as a single-line vector operation, and so is very performant.

8 Block 7: Scan Conversion - Cometto

Scan conversion handles the task of converting the stored beam data (in our case, in Mag_image) from its polar reference (beam/sample or angle/distance) to a rectangular reference that can be used as data for an image. Additionally, in order to form a continuous image, the values between the beams must be interpolated.

8.1 Background

Before discussing the implementation, we will first discuss the information necessary to understand and accomplish this task. First, the signal data is stored in an array named `Mag_image` where each row corresponds to a beam (angle), and each column corresponds to a sample (distance). In our case, the array is 21×4000. The output of the scan conversion stage is a 100×201 pixel image, which will be stored as a 100×201 array called `image`.

We will define the pixel at index $[1, 101]$ as our position $(x, y) = (0, 0)$. From there, we can assign each pixel an angle, θ_p, and a distance, r_p. Then, we can perform bilinear interpolation using the equation

$$\begin{aligned}\texttt{image[r,c]} = (1-\beta)\left((1-\alpha)\texttt{Mag\_image[k,n]} + (\alpha)\texttt{Mag\_image[k+1,n]}\right) \\ +(\beta)\left((1-\alpha)\texttt{Mag\_image[k,n+1]} + (\alpha)\texttt{Mag\_image[k+1,n+1]}\right)\end{aligned}$$

with

$$\alpha = \frac{\theta_p - \theta_k}{\theta_{k+1} - \theta_k}$$

and

$$\beta = \frac{r_p - r_n}{r_{n+1} - r_n}$$

as the fractional offsets in the angle and distance directions, respectively. This allows us to use use a weighted average of the four nearest known values to calculate the value at every pixel of the output image.

8.2 Implementation

In order to implement this algorithm in a fast way, we need to precompute as much as possible. With the right values precomputed, it is possible to completely vectorize the function, as is done in `scan_conversion.m`. Ultimately, we need 8 precomputed values, described in Table 4.

Parameter	Description
`ind_bkn`	Set of linear indices for the (k, n) value for each pixel
`ind_bk1n`	Set of linear indices for the $(k+1, n)$ value for each pixel
`ind_bkn1`	Set of linear indices for the $(n, k+1)$ value for each pixel
`ind_bk1n1`	Set of linear indices for the $(n+1, k+1)$ value for each pixel
`BMAM`	The value of $(1-\beta)(1-\alpha)$ for each pixel
`BMA`	The value of $(1-\beta)(\alpha)$ for each pixel
`BAM`	The value of $(\beta)(1-\alpha)$ for each pixel
`BA`	The value of $(\beta)(\alpha)$ for each pixel

Table 4: Precomputed Values for Scan Conversion

To do these precalculations, we first need the distance and angle of each pixel. To begin, we can calculate that the distance of each sample n is given by

$$r_n = \frac{nC}{2F_s}$$

with $C = 1136$ feet/s and sampling frequency F_s. We can also find that the angle of each beam k is given by

$$\theta_k = \arcsin(\frac{2kf_c}{F_s})$$

for pulse frequency f_c.

With these, we can calculate the pixels per foot in the row (y) direction with

$$\texttt{rppf} = \frac{\texttt{image\_rows} - 1}{r_m ax}$$

and in the column (x) direction with

$$\texttt{cppf} = \frac{\texttt{image\_columns/2} - 1}{r_m ax}$$

Note here that a 100×201 pixel image has a ratio $\frac{\texttt{rppf}}{\texttt{cppf}}$ that is very close to one, which means that the output image is not distorted.

Finally, we can find the rectangular coordinates (p_x, p_y) of the pixel at index $[p_r, p_c]$. Noting that $(0, 0)$ is at $[1, 101]$ with

$$p_x = \frac{(pc - 101)}{\texttt{cppf}}$$

and

$$p_y = \frac{(pc - 1)}{\texttt{rppf}}.$$

Then, we can find the distance and angle of each pixel with

$$r_p = \sqrt{p_x^2 + p_y^2}$$

and

$$\theta_p = \arctan(\frac{p_x}{p_y}).$$

Finally, we can find the necessary indices. The fractional indices of the pixel can be found with

$$k_p = \frac{F_s \sin(\theta_p)}{2f_c}$$

and

$$n_p = \frac{2F_s r_p}{C}.$$

Then, we can round n_p down (`floor`) for n, and add one for $n+1$. Similarly, rounding k_p down yields k, and adding one yields $k+1$. Thus, we have the indices for our four neighboring points. The last step is to convert $(row, column)$ indices to linear indices, which is done using the `sub2ind` script in MATLAB.

Our second precomputational task is to find the fractional offset between the neighboring points. The fractional offset in the θ direction is given by

$$\alpha = \frac{\theta_p - \theta_k}{\theta_{k+1} - \theta_k},$$

and we can calculate the fractional offset in the r direction with

$$\beta = \frac{r_p - r_n}{r_{n+1} - r_n}.$$

Then, in order to vectorize the code, we can compute the four scaling factors for bilinear interpolation,

$$\begin{aligned}(1-\beta)(1-\alpha)\\(1-\beta)(\alpha)\\(\beta)(1-\alpha)\\(\beta)(\alpha),\end{aligned}$$

for each point. Thus, we have all 8 of the necessary values that can be precomputed. These are done in the `scan_conversion_precompute` function.

With all of these precalculations done, the calculation using the `Mag_image` data can be done very quickly. Implementing the bilinear interpolation calculation using the proper indexing, we have the simple and fast, single line that follows.

```
1    image=BMAM.*Mag_image(ind_bkn)+BMA.*Mag_image(ind_bk1n)...
2        + BAM.*Mag_image(ind_bkn1) + BA.*Mag_image(ind_bk1n1);
```

8.3 Development Process

Before the analysis, we will briefly discuss the development process. The design began by using a `for` loop to create a functional, yet not optizimied solution. This enabled many bugs to be worked out before transitioning to the vectorized version. Additionally, it laid the foundation for understanding and implementing the necessary precomputations.

8.4 Test and Analysis

In this stage, there are two main aspects that must be tested and analyzed. First, we will confirm that the precomputations are performed correctly. To do so, we will inspect Figure 24.

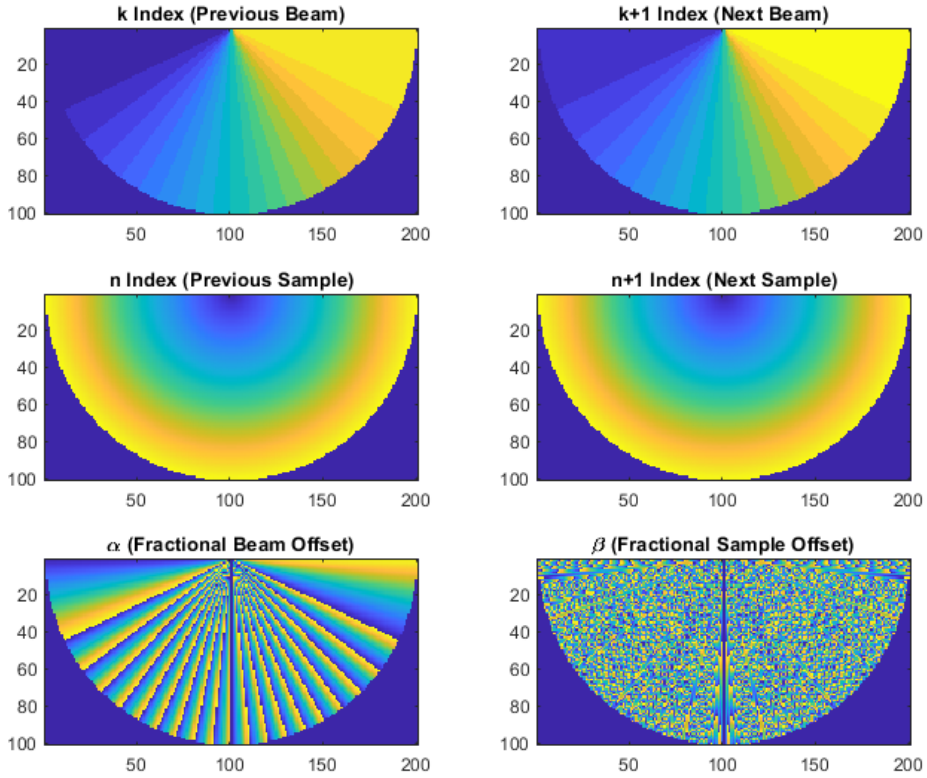

Figure 24: Precomputation Plots by Image Pixel

Notice that the data looks as expected. First, we can see the previous and next beam indices sweep across the possible angles. Additionally, the previous and next sample indices sweep through the distances. Then, the fractional beam offset increases from zero as it gets closer to the next beam index. The fractional sample offset does as well, though on a much smaller scale. Thus, the precomputations were done successfully.

Next, we have three sets of synthetic data to confirm that the interpolation performs as intended. First, we will transform and interpolate a block, as in Figure 25

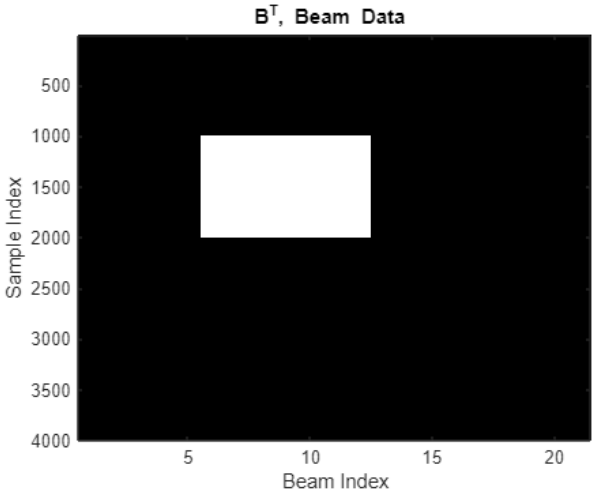

Figure 25: Beam Data, Scan Test 1

This beam data is successfully transformed, as can be seen in Figure 26.

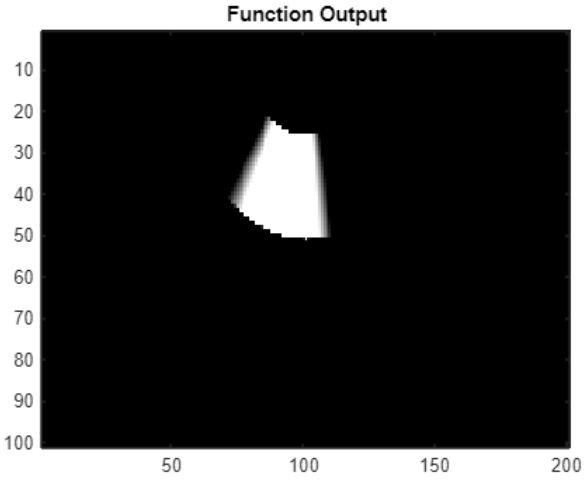

Figure 26: Converted Image, Scan Test 1

The next test, increasing in complexity, is with a double checkerboard, as can be seen in Figure 27.

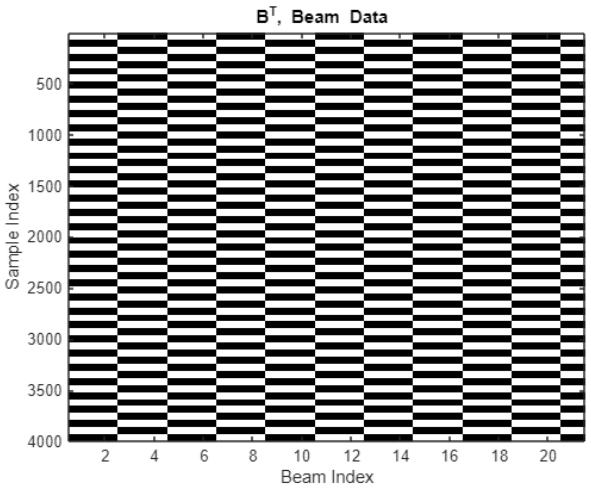

Figure 27: Beam Data, Scan Test 2

Again, this beam data is successfully transformed, as can be seen in Figure 28.

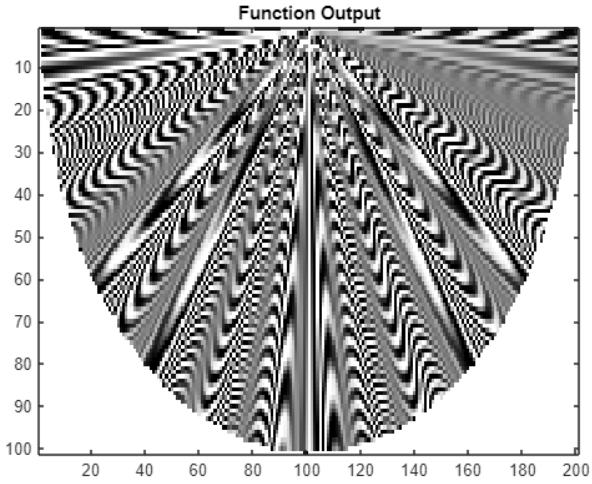

Figure 28: Converted Image, Scan Test 2

A third test is with a single, alternating checkerboard, as can be seen in Figure 29.

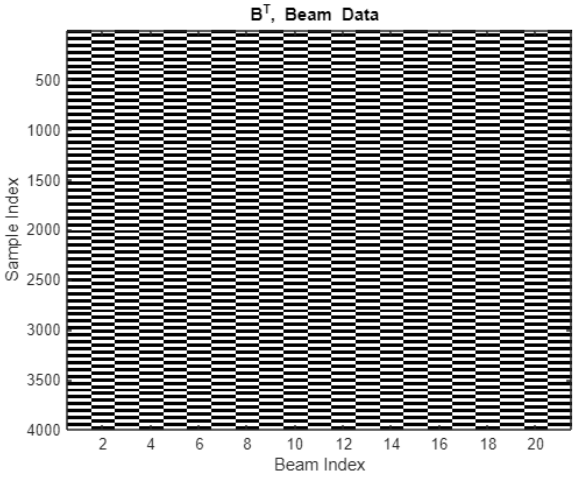

Figure 29: Beam Data, Scan Test 3

Again, this beam data is successfully transformed, as can be seen in Figure 30.

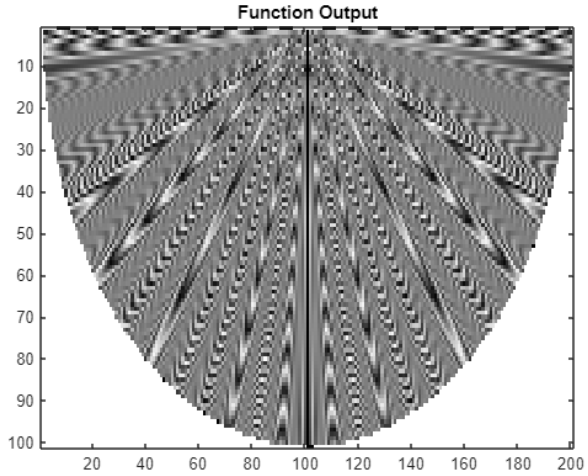

Figure 30: Converted Image, Scan Test 3

After conducting tests using synthetic data, we can further confirm using the real test data. As can be seen in Figure 31, the system successfully interpolates real data as well.

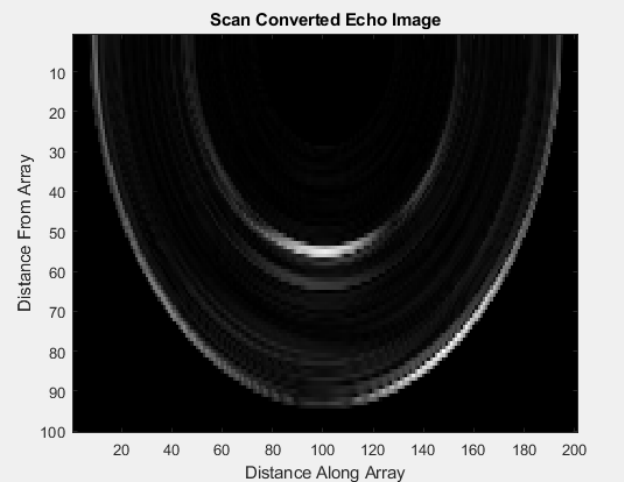

Figure 31: Converted Image, Real Test Data

The last aspect of the function to analyze is its 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 5. The same data (the double checkerboard test pattern) was used, and the computer was in as similar a state as possible. (Note, the provided `.p` function had a non-suppresed (no semicolon) line that outputted `max_pixel = 10`, which needed to be suppressed using `evalc`, likely reducing the speed of the provided function.)

Function	Time (μs)
Provided `.p`	4002
Rewritten `.m`	742.3

Table 5: Provided vs Rewritten Scan Conversion Performance

As can be seen, the rewritten and vectorized `scan_conversion.m` successfully interpolates and transforms the data into the correct format about 5 times faster than the provided function. It is able to do this do to a substantial increase in precomputations. However, these precomputations only need to be calculated if certain configuration parameters are changed. Thus, they are written to a file, and only incur precomputation time when necessary.

9 block 9

10 Block 10: Persistence - Stentiford

The persistence stage consists of an infinite impulse response filter implemented very simply by taking weighted averages of the current image and the previous image. The resulting image is then saved to be used as the previous image in the next iteration. The relative weights of the current and previous image determine how much previous images stick around. A higher persistence, meaning more weight is given to the previous image, results in better temporal noise filtering but blurs moving objects, while the lower persistence more clearly displays moving objects and the expense of more noise grain in the image.

$$I_{t=n} = (1-p)I_{new} + p * I_{t=n-1}$$

11 Block 11: Tracking/Velocity Estimation - Csicsila

11.1 Tracking

The tracking feature add quality to the sonar project. Using an 'X' crosshair, this stage successfully follows the most prominent object as it moves across the monitor. To achieve this, the xcorr2 function was used. This function takes the signal and a template and then runs a Cross-correlation algorithm to find which part of the graph most closely matches the template. This is similar to convolution. Once that is complete, the maximum point of the correlation is found and indexed to get the row and col, to overlay the crosshair on the image.

The template is comprised of a 4x4 array filled with 1's. This template was chosen with the target in mind. The target was essentially a blob with higher values toward the center. Therefore, the template would correlate with the highest in the middle of that blob. This template plan also worked for the second object that was much longer and thinner than the first object. This is because the middle of that object still had a concentration of higher values which matched with the template.

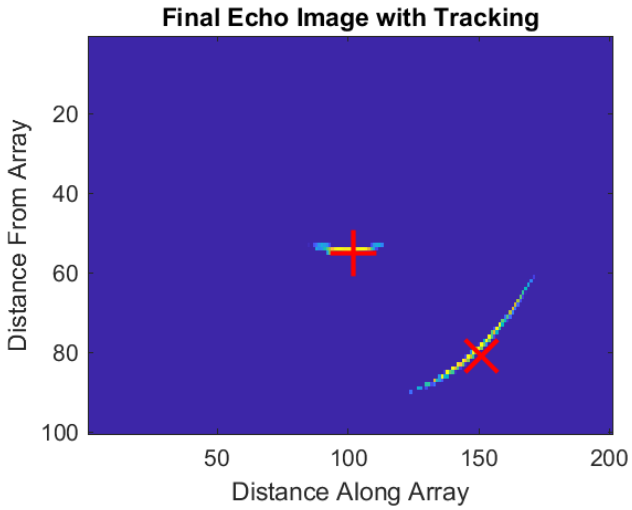

Figure 32: Final Echo Image with Tracking

Figure 32 shows the one frame where both objects are in sight with the crosshairs overlaid. The picture was colored to show the intensity around the blob better. This image also show cases the programs ability to detect two prominent images at once which will be discussed later on.

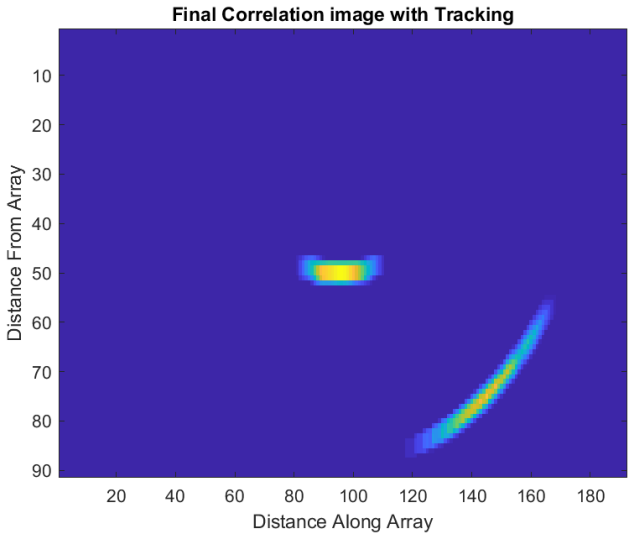

Figure 33: Final Correlation image with Tracking

Figure 33 looks very similar but shows the correlation between the template and the original signal. As you can see, the correlation made each signal thicker and more prominent. This allowed the function to better understand where the denser part of the blob was for tracking purposes.

Before testing with the live image, test data was made to ensure that it would perform correctly. The first test data utilized an array with multiple different hotspots of intensity/

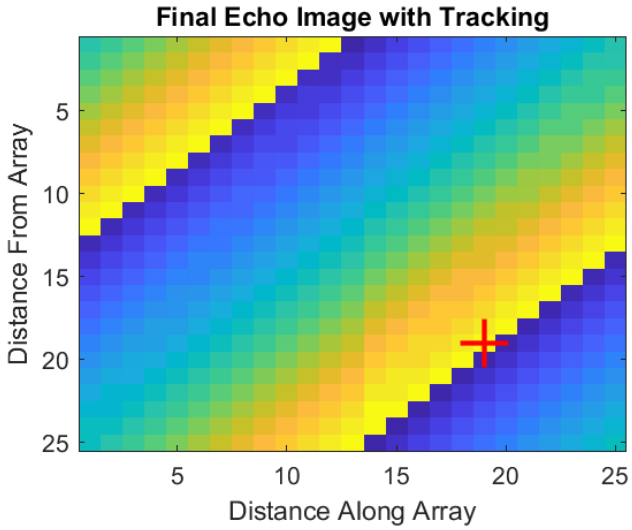

Figure 34: Test Image with Tracking

Figure 34 shows the random data with hotspots (highlighted by the lighter colors). Look what happens when the correlation with the same template is used.

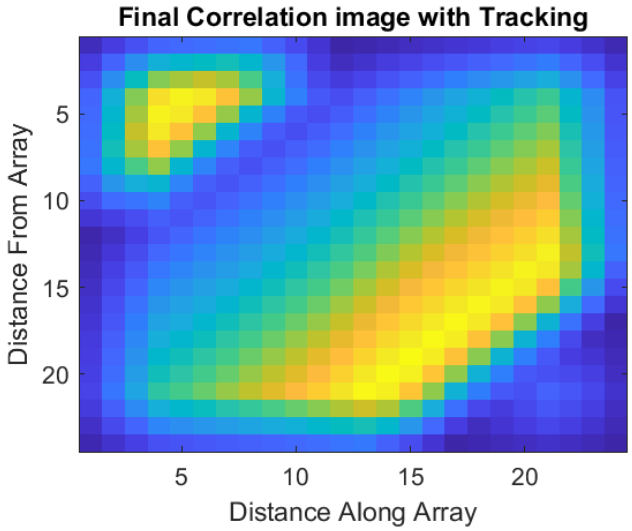

Figure 35: Test Image Correlation with Tracking

As seen in Figure 35, the correlation was able to pick out the lower right corner as being more similar to the template. This mainly due to the lower pixel count of this image. Nevertheless, the program would work with the live image.

The next task was to try to implement tracking for two objects. This was a bit more challenging and had mild success with the live image. The problem trying to be solved was finding local maxima for the data. The first theoretical approach was to use derivatives, however this would be impractical and very slow for the size of this data. Therefore, another method was implemented. First, correlation was done in the same manner was the first tests. Then, an offset was used to zero out the values around the max value. This way, correlation could be done again without selecting the same point.

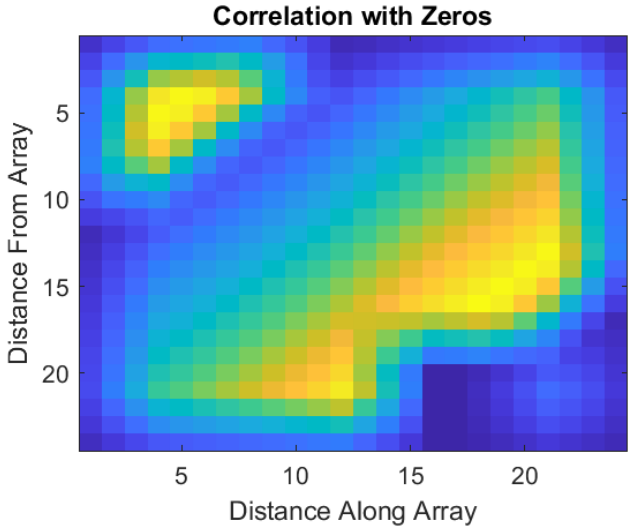

Figure 36: Test Image Correlation with Zeros

Figure 36 shows the correlation after the zeros are removed. Comparing this to 35, there is clearly an empty

spot. The zero offset could be increased to essential increase the radius in which to ignore objects around the original object.

However, an issue arose with edge cases. When setting the zeros with the offset it would sometimes set values outside the range of the array to 0 which would throw an error. To fix this, the edges of the scan were temporarily removed to avoid the error. Then the location was calculated then the edges were returned. This did not affect quality, and solved the issue.

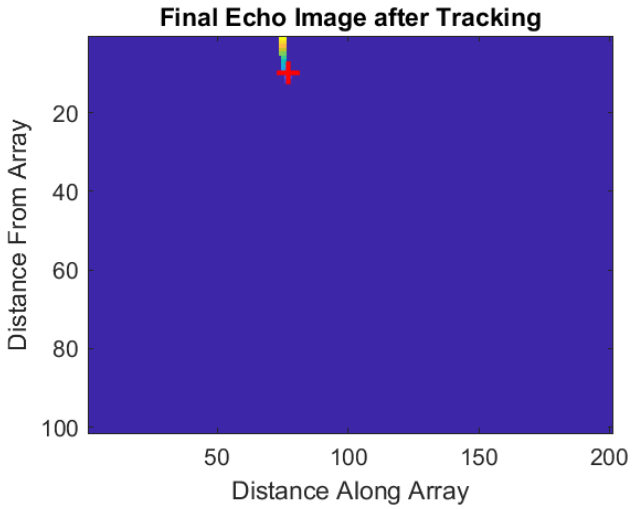

Figure 37: Image Correlation Edge Case

Figure 37 was created to test the edge cases so prove that the program could still detect images close to the edge.

11.2 Velocity

Velocity was extremely simple to produce. Because the tracking showed the location of the image, a simple distance equation could be used.

$$d = \sqrt{(pixel_per_foot_row(x_2 - x_1))^2 + pixel_per_foot_col((y_2 - y_1))^2} \tag{2}$$

The last tricky part was figuring out the timing to get the velocity. Knowing that each scan consisted of 4000 samples, and the new up sampled sampling frequency was 200kHz, the time was found. The final step was choosing two tracked images to get the distance. An image a iteration 5 and 50 were chosen. Therefore, the final time had to be multiplied by 45 to account for the time difference between iteration.

The estimated velocity was around 2.4 ft/s.

12 Conclusion

We successfully implemented the first five stages of the sonar system. The final image produced matches the reference image provided with the project materials. We rearranged the stages somewhat by moving bias removal into stage 3 (band limiting and denoising) and rearranging the stages so that stage 3 came first, followed by stage 1, stage 2, stage 4, and stage 5. The purpose of that was to allow for a simpler, more reliable method of implementing the required functionality. Overall, we aimed for quality first, then speed, which is reflected in our manner of implementation in code. We used vector operations where possible and minimized unnecessary memory copies or needless recalculations.

The primary challenge we faced was, in fact, not even in the actual work in MATLAB. Instead, a bigger source of issues was the Git version control system, which caused conflicts and even odd glitches in earlier drafts of this report. The lesson from that experience is that often times, it is not the actual engineering and building which causes issues, but all the things surrounding and supporting those.

Documentation

We worked as a team on this project. Additionally, we 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 at this link. We also used that same ChatGPT conversation to discuss the mathematics behind accounting for time gain compensation.