Fractional Delay Interpolation and Beamforming

Brad Hundl

Abstract

This report describes the underlying principles of Fractional Delay Interpolation and Beamforming. The report contains the analysis of simulations created through MATLAB. Our goal is to form a strong intuition for the theory behind these methods and gain experience by applying them to applicable cases. During the analysis we attempt to draw conclusions on how useful these methods can be in a practical setting and what cases they might best be applied to.

I. Introduction

In signal processing, Signal to Noise Ratio (SNR), is a highly discussed topic. In many devices today this ratio which compares the level of signal power to the level of noise power is attempted to be maximized for the best performance possible. Beamforming is a method that can be used to help maximize SNR. This technique involves the outgoing signal to be sent out in a particular direction while the receiving antenna array is outfitted with many sensors spaced in such a way that the direction that the signals arrive from is critical to processing the signal correctly. This direction that the signal comes from, in relation to the antenna array is called the Angle of Arrival (AOA).

An incoming signal contacts the antenna elements at different points in time which can be taken advantage of since there is a particular direction in which the signal that we want to receive, and process is coming from. In this setup, stray signals coming from directions that are outside that of the expected AOA, will be deconstructive in amplitude when processed while signals coming from our expected AOA will experience a gain in amplitude during the processing phase.

This process can also work in the reverse order. For example, if we were looking for a particular signal, we can create a signal processing program that applies beamforming but for a wide range of possible AOAs. This can be a useful method for discovering the signal you are looking for and allows one to learn the direction from which an incoming signal is coming from even though there may be many other signals causing unwanted noise.

II. Interpolation Filter

A. *Background*

Continuous time signals can be created by using discrete-time signals using sampling. Given a continuous time signal, the discrete time signal can be expressed as the following, where **T** is time in seconds between each sample:

$$x[n] = x(nT) \quad (1)$$

When it comes to reconstructing a signal based on samples, the question of how exactly to do it arises. The reconstruction formula for a signal is given below:

$$x(t) = \sum_{k=-\infty}^{\infty} x(kT)\frac{\sin[\pi F_s(t-kT)]}{\pi F_s(t-kT)} = \sum_{k=-\infty}^{\infty} x[k]sinc[F_s(t-kT)] = \sum_{k=-\infty}^{\infty} x[k]sinc[tF_s - k] \quad (2)$$

The reconstruction yields an expression that is close to a discrete-time convolution of an input signal and impulse response. The equation is close and not exact because of the mixing of discrete and continuous time. The final continuous-time output signal can be expressed as:

$$x(t) = \sum_{k=-\infty}^{\infty} x[k]h(t - kT) = \sum_{k=-\infty}^{\infty} x[k]sinc[tF_s - k] \quad (3)$$

Part I of this project is concerned with delaying the discrete-time sequence, rather than the reconstruction of the signal to make it easier. Reconstructing the continuous time signal can be thought of as equivalent to delaying the continuous time signal and resampling the signal after. The following equation represents the delayed signal:

$$y(t) = x(t - t_0) = \sum_{k=-\infty}^{\infty} x[k]h(t - t_0 - kT) = \sum_{k=-\infty}^{\infty} x[k]sinc[(t - t_0)F_s - k] \quad (4)$$

As mentioned above, once the signal is delayed, the signal can be resampled to obtain the sequence. This will leave us with:

$$y[n] = y(nT) = \sum_{k=-\infty}^{\infty} x[k]h(nT - dT - kT) = \sum_{k=-\infty}^{\infty} x[k]sinc[n - d - k] \quad (5)$$

What we are left with can be reformatted into:

$$h[n] = sinc[n - d] \quad (6)$$

Which is the discrete-time convolution we need.

B. *MATLAB Implementation of FDI*

To implement Fractional Delay Interpolation, we used the following process: To begin, we construct an impulse response in such a way that allows us to delay an incoming signal. The relationship between this impulse response and the input signal comes from the fact that we convolve both signals together to receive an output signal that matches the input signal, only this time delayed by the desired amount. FDI is particularly important for delaying discrete-time input signals by a non-integer amount, hence the term “Fractional Delay”. The interpolation part of the term comes from the idea that we want to interpolate values for points in time which we do not actually possess since we are dealing with discrete-time signals.

To perform a fractional Delay on a particular incoming discrete-time signal, we construct the Interpolation Filter. Our Interpolation Filter is created by multiplying two functions together. One of these is a Sinc function where the domain on which it is constructed is shifted by the desired shift we label as the variable, **d**. We then create a window function that can be any type of window function we choose and assign it a length that matches that of the Sinc function. Once we have both functions, we multiply each of the corresponding elements together. The window function allows us to apply a sort of precedence to the terms closest to the origin and disregard those which are far away from the origin. We would like to note that we are referencing the origin about which the Sinc function is at maximum amplitude, as this point on the domain that is used to allow for the proper delay.

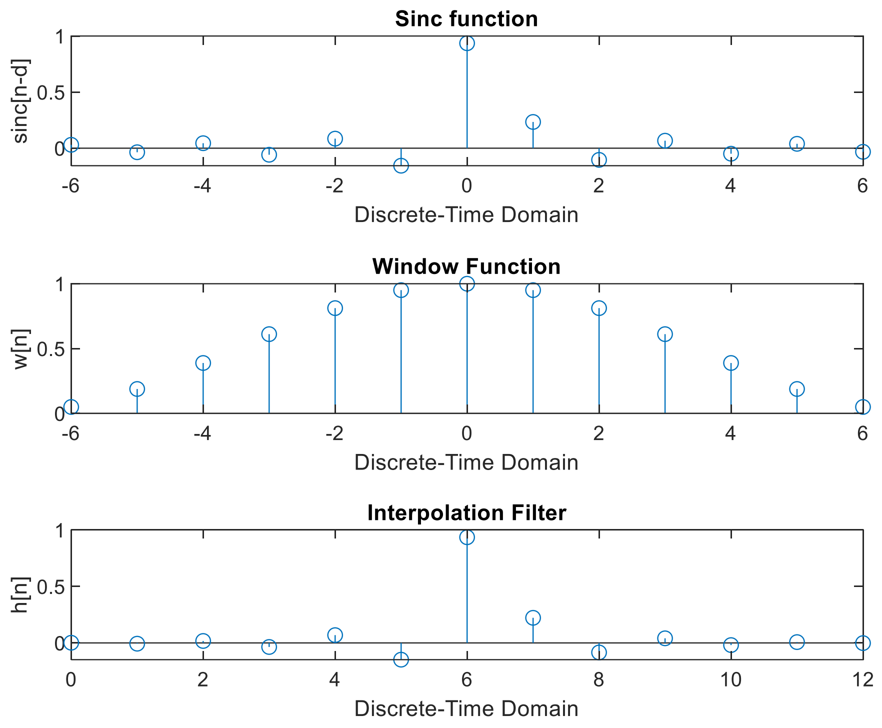


Figure 1.) This Figure shows both the Sinc and Window function, followed by the resulting plot of their multiplication.

Once we constructed a working interpolation filter, we then began using it to apply delays to a few discrete-time signals. The first signal we used as an input was a Chirp Signal. This chirp signal had a Beta equal to 0.1, a length of 150, and a y-value equal to Beta/length. This signal can be seen below.

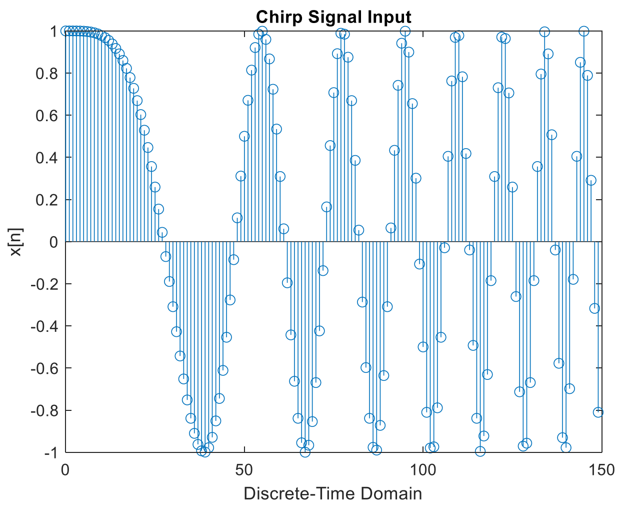


Figure 2.) This figure contains the plot of the Chirp Signal used in Part 1 of our analysis.

With an interpolation filter ready for application and an input signal in hand, we were then able to convolve the two together in MATLAB using the "Conv" function which applied a successful delay as can be seen in the next figure.

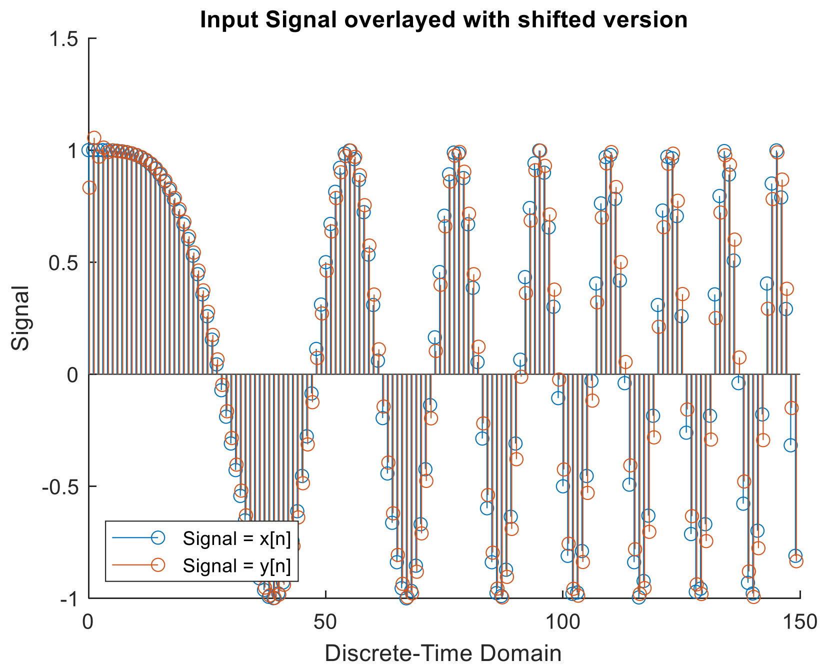


Figure 3.) This figure shows both the delayed and non-delayed input signal after applying the interpolation filter on the chirp signal. Delay = 0.2 samples.

Before drawing conclusions about our interpolation filter, we attempted to apply it to a different input signal for comparison. This time we used a discrete-time Sin function which can be seen in the figure below.

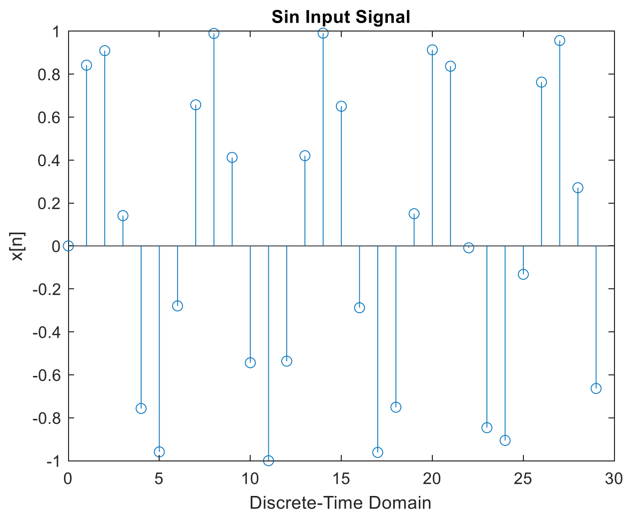


Figure 4). This figure shows the Sin function used in Part 1 analysis.

Once again, applying the same basic principles we constructed an interpolation filter to be used as our impulse response (h[n]), as we showed previously in **Figure 1).** and convolved the h[n] with our Sin function from **Figure 4).** which gave us the following figure.

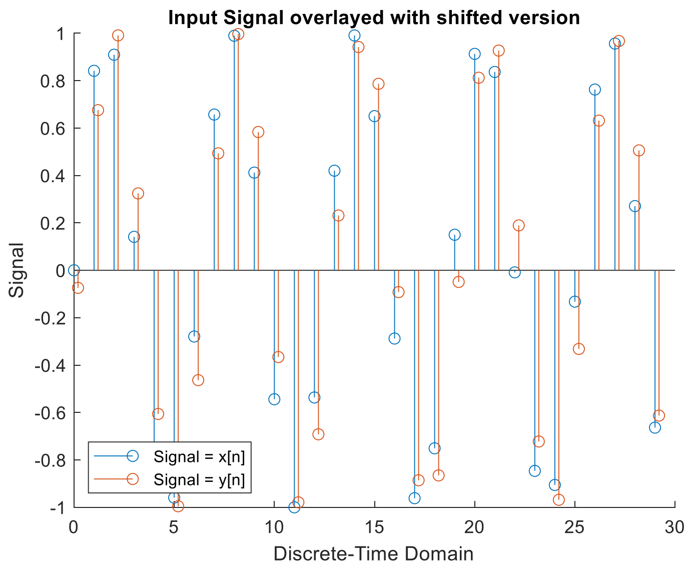


Figure 5.) This figure contains both the original and delayed versions of the Sin input signal. Delay = 0.2 samples.

III. Interpolation Filter Results:

To begin our analysis of our interpolation filter, we will look at both input cases. Right off the back, we noticed the discontinuity that occurs at the beginning of the signal. This is to be expected because of the means in which the delayed signal is interpolated. Convolution can cause slight errors here because the signals are not yet completely overlapping during this interval in the convolution. More importantly, however, there is a noticeable trend that is especially apparent when looking at both cases together. As the magnitude of difference in the output values of adjacent samples increases, so does the error in the interpolated values.

For example, if we look at the beginning of the Chirp signal's delay when there is little variance in the amplitude of x[n], the values for x[n-d] are interpolated to a high level of accuracy. As the function begins to increase in frequency, while maintaining amplitude, there is high variation from point to point of x[n]. This results in interpolated values at those sample points which do not appear to be as accurate as the previously interpolated values. The reason for including the Sin function in our analysis is to verify this trend. The Sin function was generated with a relatively high frequency making it more difficult for our filter to interpolate the delayed values of the function.

IV. Application of FDI Through Beamforming

A. *Background*

As mentioned in the introduction, Beamforming involves an incoming signal with a specific angle of arrival and an antenna array to intercept the signal at many different points on the antenna. The major issue that must be addressed and that is ultimately the key to beamforming's success is that each of the elements/sensors on the antenna array will receive the incoming signal at a different point in time. The solution to this is to implement the previously discussed method of FDI. We can use FDI in order to delay the samples received by each of the antenna elements so that when delayed the proper amount and then summed together, we are able to get the desired signal. Utilizing the geometry of our antenna array, the spacing of elements, the angle of arrival, and the speed of light, we are able to derive a formula that gives us an accurate estimation of how long to delay the element's samples.

Because we are using FDI, we are able to delay discrete-time signals by any amount of time, fractional or not, that we receive from our delay formula. We can add a division to obtain the sample shift amount and apply that shift to the corresponding data column.

B. *MATLAB Implementation of Beamforming*

We will now attempt to apply Beamforming to the following scenario in order to analyze simulation results and see if the theory holds up. In this case we look at an antenna array that has elements spaced linearly apart with a length, d_e. This specific antenna array contains a total of 64 elements. We represent each specific element as N_e with N being the element number from 1 to 64. Originally, for this case, we had no direct information on which direction the input signal was coming from. We only had a window of 90 degrees to work with. In this implementation, our program processes the signal for any possible incoming direction from -45 to +45 degrees in relation to our antenna array. As can be seen below, we can obtain a delay equation for each element using the following formula, $N_e d_e sin(\theta)$.

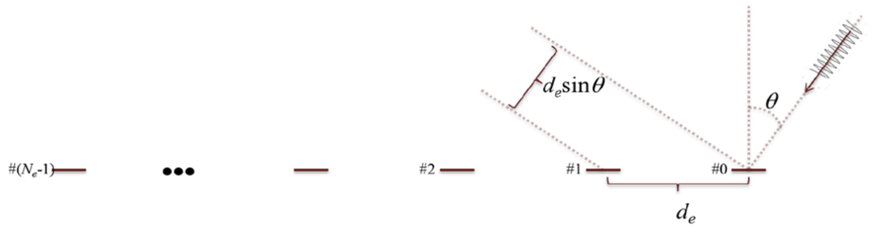


Figure 6.) This figure shows the antenna array and how the input signal is being intercepted by the antenna elements.

In this example, we look at a csv file with a 600x64 data set the represents an incoming chirp signal. Each of the 64 columns represent each of the 64 elements on the antenna. Every element has 600 sample points representing the signal that each specific element received from the signal source. In order to make sense of the signal we had to apply the proper delay to each of the elements so that signals could be summed up to produce the signal we are looking to receive.

We know that whichever antenna element that receives the signal first must be delayed the longest until the element that receives the signal last is ready to be processed with the rest of the elements. We also know that the signal could be coming at any angle from -45 to +45 degrees which means we needed to carry out our signal processing for each possible angle of degrees and that way we could use the resulting matrix to tell what the AOA is based on which angle returns the signal with the highest amplitude. From that point on, we would then only need to process the incoming signal for the specific AOA we find.

To implement this in MATLAB we first created a function that could be used by our program to modularly calculate the delay that needed to be applied to each of the elements. Once we implemented this, we used a pair of nested for loops. In the interior loop we need to keep track of an index for whichever element we are delaying, as well as an N_e constant for use in the equation to apply the right delay amount. After each element was delayed properly, the data for each element could be added up for the specific AOA. The result was added to a 600x64 matrix as a new column before incrementing to the next possible AOA. Using the intensity plot below we are able to see for what AOA the magnitude of the signal is greatest.

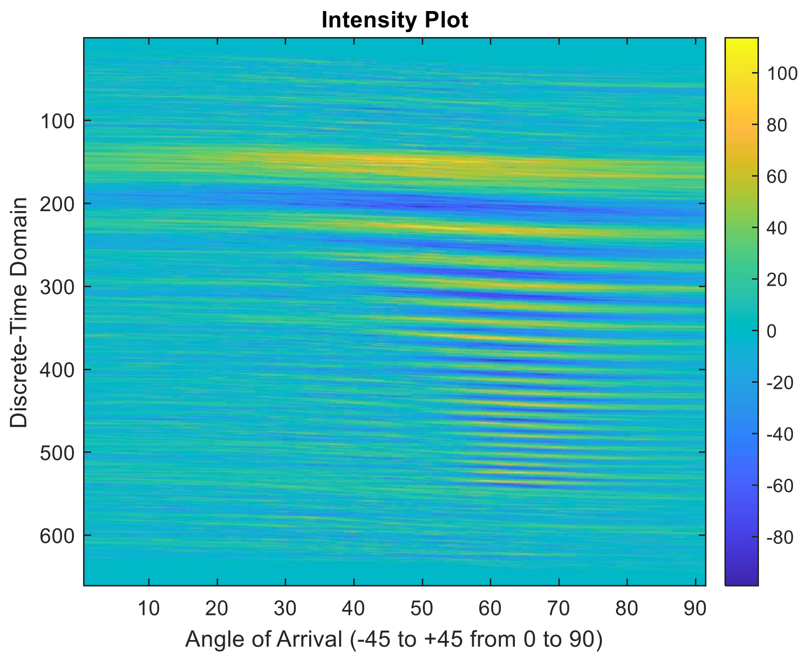


Figure 7.) Above is an intensity plot that displays intensity based on angle of arrival over 600 samples.

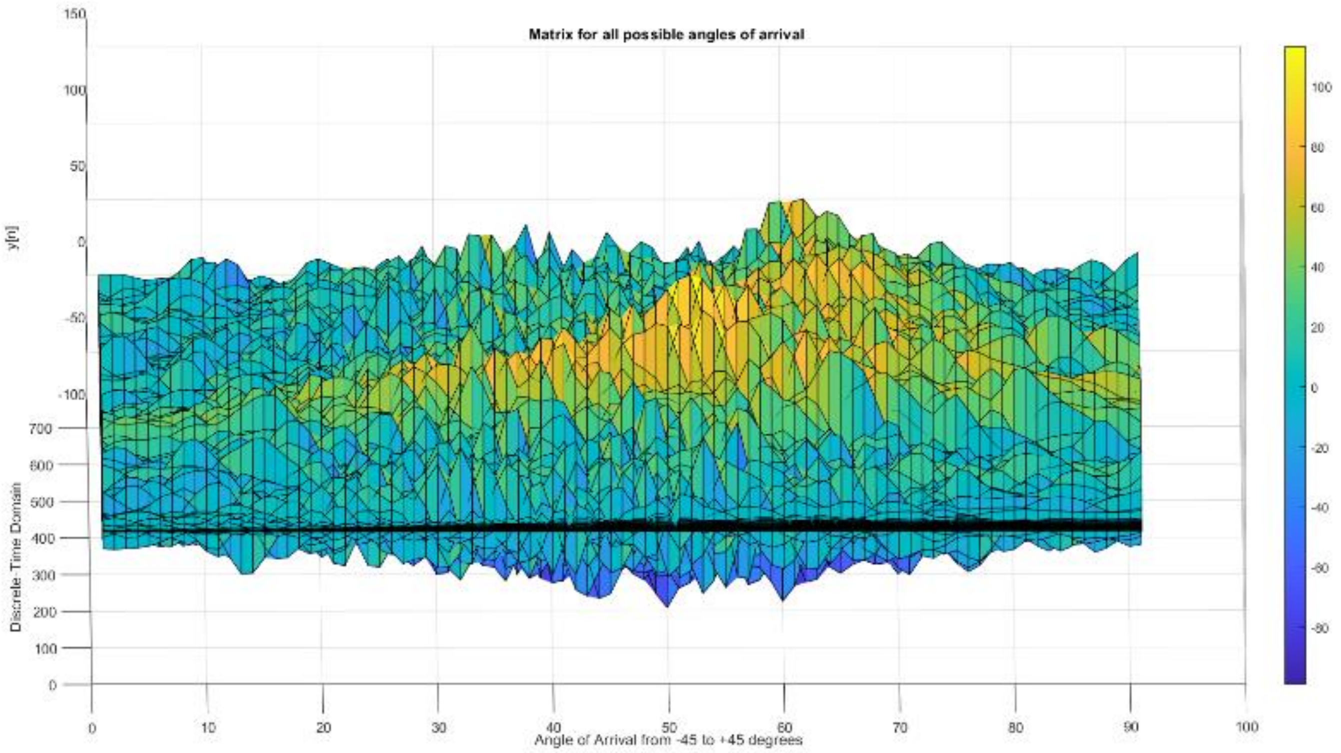
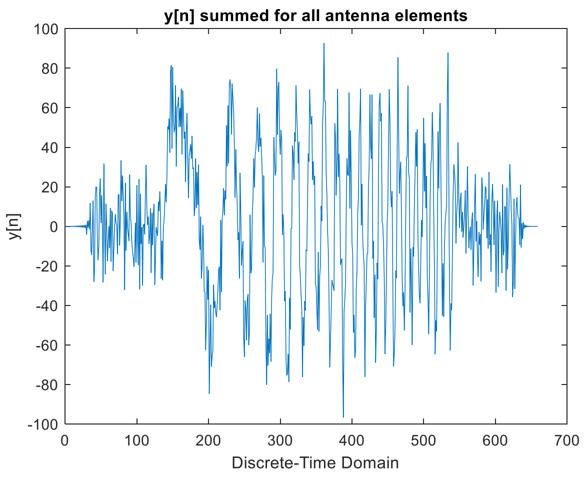


Figure 8.) Pictured above is the entire Matrix graphed in a 3-D intensity plot.

We can observe that the amplitude of the signal we are wanting to receive is greatest around an AOA of +10 to +20 degrees which allows us to estimate the AOA. To further emphasize how important, it is to sum the antenna elements for the correct angle of arrival we can refer to the plot below.



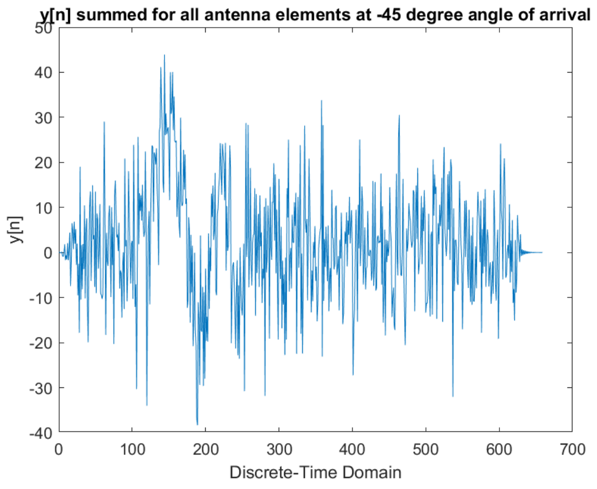


Figure 9.) The Above plots show the signal constructed from the sum of all antenna elements for a +15 degree angle of arrival on the right and a -45 degree angle of arrival on the left side.

As we can see, when using the correct angle of arrival, we can clearly make out a chirp signal which is what was originally sent. On the left, when using the incorrect angle of arrival, the elements aren't delayed by the correct amount of samples before being summed together, and therefore we end up with a great deal of noise in the signal.

There is another observation to make when looking at the 3-D plot from a side view as well which can be seen below in **Figure 10.)**. We can see that the signal, if processed with the correct sample delay is also amplified by Beamforming. The amplitude of the signal is much higher than that of the original signal and especially compared to any low amplitude noise. This results in an increase in the SNR which aligns with what we believed should happen.

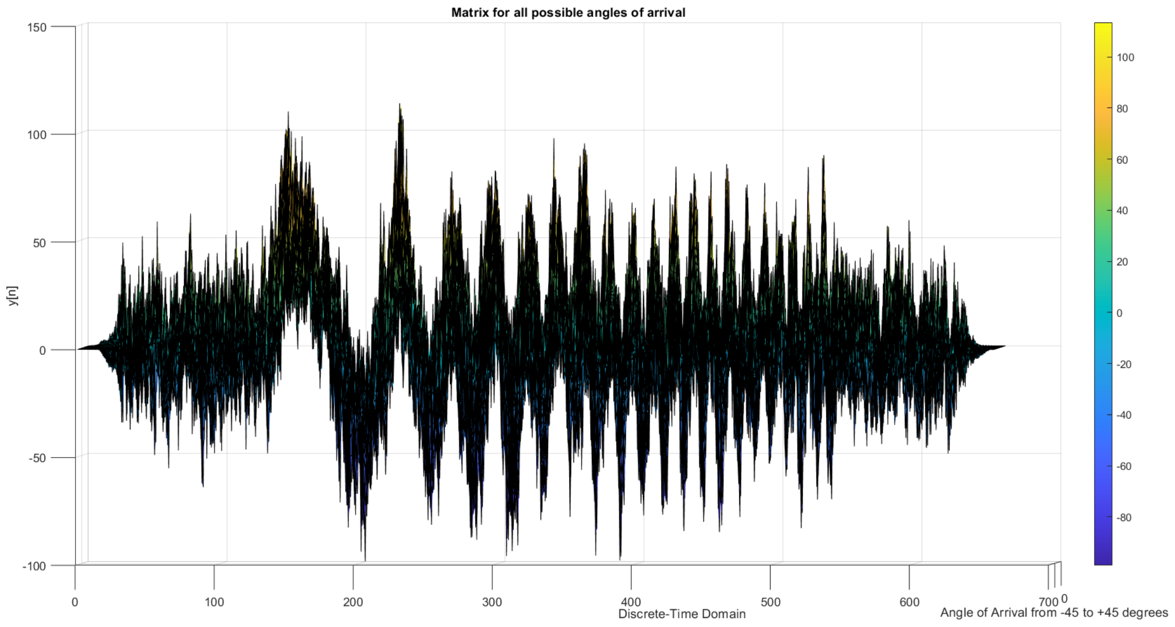


Figure 10.) Side view of 3-D Surf plot that displays amplification of original chirp signal.

V. Final Summary and Conclusions

We have seen through our simulations that the theory is indeed backed up by the example where we applied beamforming to the chirp signal dataset. While there is still much testing to be done, we can also note from the original analysis of FDI in generally, it is not as accurate when it comes to high frequency, oscillating signals. At the end of the report, we were told that the actual angle of arrival for the dataset was 20 degrees which was included into the range that we estimated based on our intensity plot. Overall, Beamforming is definitely a topic that should continue to be researched on and optimized as we have seen that it can be helpful in increasing SNR which can always stand to be further improved.

VI. Appendix

A. Part1Chirp.m

```
clc;
clear all;
close all;

% Declaring constants for x[n].
Lx = 150;
B = 0.1;
y = B/Lx;

% Declaring x-axis for variable n.
N0 = 150;
n = (0:(N0-1))';

% Creates a chirp signal called x[n].
x_n = cos(pi.*y.*(n.^2))

% d is the non-integer signal delay.
d = .2;

% Creating the horizontal axis for h[n]
% N1 = 6;
N1 = 6;
n_h = (-(N1):(N1))';

% n_d is the shifted version of the x-axis by the delay.
n_d = minus(n_h, d);

% Creating sinc function with respect to n_d.
%sinc_n_d = sin(n_d)./n_d;
sinc_n_d = sinc(n_d);

% Creating a window function.
%w = blackman(N1*2);
w = hanning((N1*2)+1);

% Multiplying shifted sinc function by the window function.
h_n = sinc_n_d.*(w);


% Convolving delayed h[n] with x[n]
y = conv(h_n, x_n);

% length of the axis for graphing h[n]
h_nAxis = (0:2*N1);

% i think the plots are off by half the length of h[n] so move y
over to
% left

figure(1)
stem(n, x_n);
title('Chirp Signal Input')
xlabel('Discrete-Time Domain')
ylabel('x[n]')

figure(2)
tiledlayout(3,1)

nexttile
stem(n_h, sinc_n_d);
title('Sinc function')
xlabel('Discrete-Time Domain')
ylabel('sinc[n-d]')

nexttile
stem(n_h, w);
title('Window Function')
xlabel('Discrete-Time Domain')
ylabel('w[n]')

nexttile
stem(h_nAxis, h_n);
title('Interpolation Filter')
xlabel('Discrete-Time Domain')
ylabel('h[n]')

figure(3)
hold on;
stem(n, x_n);

y_n = 0:149;
y_n = minus(y_n, -d);
y = y(7:156);
stem(y_n, y);

title('Input Signal overlayed with shifted version')
xlabel('Discrete-Time Domain')
ylabel('Signal')
legend({'Signal = x[n]','Signal = y[n]'},'Location','southwest')

hold off;
```

B. Part1Sin.m

```
  clc;
clear all;
close all;

% Declaring x-axis for variable n.
N0 = 30;
n = (0:(N0-1))';

% Creates a sin signal called x[n].
x_n = sin(n);

% d is the non-integer signal delay.
d = .2;

% Creating the horizontal axis for h[n]
% N1 = 6;
N1 = 6;
n_h = (-(N1):(N1))';

% n_d is the shifted version of the x-axis by the delay.
n_d = minus(n_h, d);

% Creating sinc function with respect to n_d.
%sinc_n_d = sin(n_d)./n_d;
sinc_n_d = sinc(n_d);

% Creating a window function.
%w = blackman(N1*2);
w = hanning((N1*2)+1);

% Multiplying shifted sinc function by the window function.
h_n = sinc_n_d.*(w);

% Convolving delayed h[n] with x[n]
y = conv(h_n, x_n);

% length of the axis for graphing h[n]
h_nAxis = (0:2*N1);

% i think the plots are off by half the length of h[n] so move y
over to
% left
```

```
figure(1)
stem(n, x_n);
title('Sin Input Signal')
xlabel('Discrete-Time Domain')
ylabel('x[n]')

figure(2)
tiledlayout(3,1)

nexttile
stem(n_h, sinc_n_d);
title('Sinc Function')
xlabel('Discrete-Time Domain')
ylabel('sinc[n-d]')

nexttile
stem(n_h, w);
title('Wndow Function')
xlabel('Discrete-Time Domain')
ylabel('w[n]')

nexttile
stem(h_nAxis, h_n);
title('Interpolation Filter')
xlabel('Discrete-Time Domain')
ylabel('h[n]')

figure(3)
hold on;
stem(n, x_n);

y_n = 0:29;
y_n = minus(y_n, -d);
y = y(7:36);
stem(y_n, y);

title('Input Signal overlayed with shifted version')
xlabel('Discrete-Time Domain')
ylabel('Signal')
legend({'Signal = x[n]','Signal = y[n]'},'Location','southwest')

hold off;
```

C. FDI.m

```
function y = FDI(x_n,d)

% creates horizontal axis for impulse
response
Lh = 30;
n_h = (-(Lh):(Lh))';

% applying delay to horizontal axis for
impulse response
n_d = minus(n_h, d);

% creating sinc function with respect to
delayed axis
sinc_n_d = sinc(n_d);

% creating a window function
% w = blackman((Lh*2)+1);
w = hanning((Lh*2)+1);

% Multiplying shifted sinc function by the
window function.
h_n = sinc_n_d.*(w);

% Convolving the input signal with the
interpolation filter.
y = conv(h_n, x_n);

end
```

D. Part2.m

```
clc;
clear all;
close all;

x = load('CA1_Xs.mat');
A = struct2array(x);

c = 299792458;
de = 0.2;
T = 0.000000001;

element = 1;
yOutput = 0;
yMatrix = zeros(660, 90);
angleIndex = 1;

for theta = -45:1:45
    yOutput = 0;
    for Ne = 63:-1:0

        x_n = A(:,element);
        propDelay = Ne*de*sind(theta)/c/T;

        currY = FDI(x_n, propDelay);
        yOutput = yOutput + currY;

        element = element + 1;
    end

    yMatrix(:,angleIndex) = yOutput;

    angleIndex = angleIndex + 1;
    element = 1;

end


figure(1);
imagesc(yMatrix);
title('Intensity Plot')
xlabel('Discrete-Time Domain')
ylabel('Signal')
colorbar;

figure(2);
surf(yMatrix);
title('Matrix for all possible angles of arrival')
xlabel('Angle of Arrival from -45 to +45 degrees')
ylabel('Discrete-Time Domain')
zlabel('y[n]')
colorbar;
```

E. TestingFile.m

```
clc;
clear all;
close all;

x = load('CA1_Xs.mat');
A = struct2array(x);

c = 299792458;
de = 0.2;
T = 0.000000001;
```

```
element = 1;
ySum = 0;

theta = 20;

for Ne = 63:-1:0

    x_n = A(:,element);
    propDelay = Ne*de*sind(theta)/c/T;
    currY = FDI(x_n, propDelay);
        ySum = ySum + currY;

        element = element + 1;
    end

y_n = 0:659;
figure(6);
plot(y_n, ySum);
title('y[n] summed for all antenna elements at a 20 degree
angle of arrival')
xlabel('Discrete-Time Domain')
ylabel('y[n]')
```