Naval Physical and Oceanographic Laboratory, DRDO

Internship Report

Beamforming for Uniform Linear Arrays

Author:
S.V. Rajendran
CET-ID: [redacted]
Sophomore
College of Engineering
Trivandrum

Direct Report:
[redacted]
Scientist-F
Naval Physical and
Oceanographic Laboratory

Mentor:
[redacted]
Junior Research Fellow
Indian Institute Of Science

NAVAL PHYSICAL AND OCEANOGRAPHIC LABORATORY, DRDO

Abstract

Internship Report

Beamforming for Uniform Linear Arrays

by S.V. Rajendran

Sonar is a class of technologies that involve illuminating an sub-marine environmen with an acoustic signal, recording the returns (echoes) and estimating task-specific characteristics of the underwater environment. For oceanographic tasks, this means sea-floor topology, movement of marine life and presence of alien bodies. For defense tasks, this means detection and neuralizing of threats.

Depending on the task, the system-configuration and parameters varies. Some technologies require the hydrophones (sub-marine microphones) arranged in a uniformly spaced linear array, called ULAs. Due to the loss of elevation-information from such configurations, some technologies require the use of uniformly spaced planar array of hydrophones. And over the years, designing application specific hydrophone arrays are not unheard of. However, uniform linear arrays are ubuiquitous, and shall be the focus of this report.

However, a uniform linear array is the norm when working with sonar. A towed array sonar is a technology that uses this ULA attached to a moving marine-vessel, and pings the surrounding as it moves. And beamforming is the primary method used to obtain useful information from the received signal.

Beamforming is a linear method of combining recorded signals to obtain spatial information in regards to the immediate environment. This method is used in technologies ranging from simple direction-of-arrival estimation all the way to more advanced synthetic-aperture-sonar implementations. We shall be sticking to topics similar to the former owing to the novice nature of this candidate. The method primarily stems from the fact that signals recorded by the sensors have some inherent delay owing to their delay in arrivals.

In this report, we deal with simulation of reception beamforming under different conditions and envionrments, with the sole intention of learning the basics and implementing beamforming.

Acknowledgements

I take this opportunity to express my gratitude to all the helping hands that have contributed to the successful completion of the internship.

I am extremely grateful to [redacted], Outstanding Scientist, and Director, NPOL. To [redacted], Scientist 'G', Chairman HRD Council, for giving me this opportunity for an internship in this prestigious organization.

I also express my heartfelt gratitude to [redacted], Scientist 'G', Group Head HRD, and P&A, [redacted], Scientist 'E', Division Head HRD and [redacted], Technical Officer 'B', Technical Officer HRD, for facilitating the project work.

I would also like to express my deep sense of respect and gratitude to my project guide [redacted], Scientist 'E', for his guidance and advice throughout my project. I am grateful to [redacted], JRF, for the learning experience and providing a friendly and educational atmosphere.

I express my wholehearted gratitude to [redacted], Scientist 'G', Division Head(SSD) and [redacted], Scientist 'G', Group Director (ES), for their help and support throughout the term of the internship.

I would also like to take this opportunity to put across my deepest gratefulness to my dear parents, whose constant encouragement and support helped me in completing the internship.

I would like to express my sincere gratitude to Dr. Jiji CV , Principal, College of Engineering Trivandrum, for his support and encouragement for this internship. I would also like to thank Dr. Ciza Thomas, Head of Dept, ECE dept, College of Engineering Trivandrum for her concern and constant encouragement for this internship. I would like to thank PG Gijy , staff advisor, for providing me with constant support, encouragement, guidance and motivation to complete my internship.

And last, but never the least, I'd like to thank all the metaphorical giants whose shoulder I've had the privilege to stand upon.

Contents

List of Figures

List of Tables

List of Abbreviations

SONAR	**SO**und **NA**vigation **A**nd **R**anging
RADAR	**RA**dio **D**etection **A**nd **R**anging
SNR	**S**ignal to **N**oise **R**atio
ULA	**U**niform **L**inear **A**rray

Physical Constants

Speed of Light	$c_0 = 2.997\,924\,58 \times 10^8\,\mathrm{m\,s^{-1}}$
Speed of Sound	$c = 1500\,\mathrm{m\,s^{-1}}$

List of Symbols

a	distance	m
P	power	W ($J\,s^{-1}$)
ω	angular frequency	rad

For Dasher,
Labrador Retriever extraordinaire.

Introduction

SONAR is a technique that uses sound propagation to navigate, communicate with, or detect objects or or under the surface of the water. Many other methods of detecting the presence of underwater targets in the sea have been investigated such as magnetic, optical signatures, electric field signatures, thermal detection (infrared) , Hydrodynamic changes (pressure) and has had a degree of success. Unfortunately, none of them has surpassed SONAR[1] despite it possessing numerous disadvantages .

There are two types of SONAR - Passive SONAR and Active SONAR .

Passive sonar listens to sound radiated by a target using a hydrophone and detects signals against the background of noise. This noise can be self made noise or ambient noise. Self noise is generated inside the receiver and ambient noise may be a combination of sound generated by waves, turbines, marine life and many more.

Active sonar uses a projector to generate a pulse of sound whose echo is received after it gets reflected by the target. This echo contains both signal and noise, so the signal has to be detected against the background noise. The range of the target is calculated by detecting the power of the received signal and thus determining the transmission loss. The transmission loss is directly related to the distance travelled by the signal. In the case of the active sonar, half of the distance travelled by the signal to get attenuated to undergo a transmission loss detected is the range.

Beamforming is a signal processing technique that originates from the design of spatial filters into pencil shaped beams to strengthen signals in a specified direction and attenuate signals from other directions. Beamforming is applicable to either radiation or reception of energy. Here, we consider the reception part. A beamformer is a processor used along with an array to provide a versatile form of spatial filtering. It essentially performs spatial filtering to separate signals that have overlapping frequency content but generate from different spatial directions .

Background

Despite its major success in air to air detection, RADAR isn't used in sub surface identification and detection. The main reason is the radio wave is an EM wave and the sea water is highly conductive and offers an attenuation of $1400\sqrt{f}$ dB/km. This essentially means that the sea water acts a short circuit to the EM energy[1].

Even if we were to use it, to get tangible results, the target should be in short range and shouldn't be completely submerged. Contemporary submarines are nuclear and has the capacity to go deep underwater for indefinite periods. Hence RADAR is useless in subsurface application and can only used along with SONAR without bringing much to the table. Hence SONAR is the dominating method/technology when it comes to underwater detection and ranging.

Beam forming or spatial filtering is the process by which an array of large number of spatially separated sensors discriminate the signal arriving from a specified direction from a combination of isotropic random noise called ambient noise and other directional signals. In the following simulations, we deal with 32 elements separated by a distance of $\frac{\lambda}{2}$, where λ is the wavelength of the frequency for which the beamformer is designed.

The assumptions under which we perform the simulations are:

1. the noise is isotropic
2. ambient noise at the sensor-to-sensor output is uncorrelated
3. the signal at the sensor-to-sensor outputs are fully correlated .

The sensor spacing in the array is decided based on two important considerations namely, coherence of noise between sensors and formation of grating lobes in the visible region. As far as isotropic noise is concerned, the spatial coherence is zero at spacing in multiples of $\frac{\lambda}{2}$ and small at all points beyond $\frac{\lambda}{2}$. To avoid grating lobe, the spacing between sensors must be less than $\frac{\lambda}{2}$. Hence $\frac{\lambda}{2}$ is chosen as the distance between two elements in the array.

We also assume that the source is far away so as a result, the wave fronts are parallel straight lines. Since the source would be at an angle relative to the axis, the wavefront reaches each element with varying delay. As a result, the output of each element will have a phase delay from each other

Using the above figure, we can calculate the corresponding delay of each element. This will help us in determining to what degree we would have to delay the element outputs to obtain all the outputs of elements in co phase. Once the outputs are made in-phase, they are added. For an M element array, the co-phase addition increases the signal power N^2 times and the uncorrelated noise power N times. Thus the SNR is enhanced by N times.

By changing the delays of the element's output, we can "steer" the setup to give the gain to signals coming from a certain direction. This is called beam steering and is one of the most attractive features of a beamformer. However, as one 'steers' a beamformer, the width of the main lobe goes on increasing because the effective length of the array decreases.

In our simulation, we create a matrix with the columns corresponding to the output of each element for a source at a certain angle. The noise is then added. This is the output of the elements in the array. This is the basic setup. The array is then manipulated or used as input for other array manipulations to obtain the solution to the problem/objective posed.

Let the source signal be $s(k)$. The output of the elements are time-shifted versions of $s(k)$. So, for an element 'i', the output signal would be

$$y(k) = s[k - \tau_i(\theta)]$$

Using the fourier transform, we get

$$Y_i(\omega, \theta) = e^{-j\omega\tau_i(\theta)} S[\omega]$$

where

- $\tau_i(\theta) = \frac{d_m \cos(\theta)}{c} F_s$
- d_m the distance between the element considered and the element where the wavefront first strikes.
- θ the angle the rays make with the array axis
- c speed of sound in the water
- F_s The sampling frequency of the hydrophones/sensors

Steering Vector

Now, we need to construct a steering vector.

$$d(\omega, \theta) = [1, e^{-j\omega\tau_2(\theta)}, e^{-j\omega\tau_3(\theta)}, e^{-j\omega\tau_4(\theta)}, ..., e^{-j\omega\tau_M(\theta)}]$$

To obtain the element output, we multiply this matrix with the signal function.

$$\mathbf{Y}(\omega,\theta) = d(\omega,\theta)S[\omega]$$

The output signal is given by

$$\begin{aligned} Z(\omega,\theta) &= \sum_{i=1}^{M} F_i^{\star}(\omega)Y_i(\omega,\theta) \\ &= F^H(\omega)Y(\omega,\theta) \end{aligned}$$

where F^H is the matrix containing the complex weights

Complex Radiation Field

The complex radiation field produced by a linear array of N passive receivers is given by

$$Z(\omega,\theta) = \frac{1}{M}\sum_{m=1}^{M} s(\omega)e^{-j(m-1)\frac{\omega d}{c}(\cos(\theta)-\cos(\phi))}$$

Chapter 1

Single-sensor signal simulation

1.1 Aim

Here, a sine wave is created and simulated to replicate the conditions in beamforming and to see the output of a single element in the array. The noise is added by processing the SNR parameter. The changes are then observed by changing SNR. The changes are observed in both time and frequency domain. The change in time domain is observed by plotting amplitude vs time. The change in frequency domain is observed by first taking fourier transform and then plotting absolute value vs frequency.

1.2 Plots

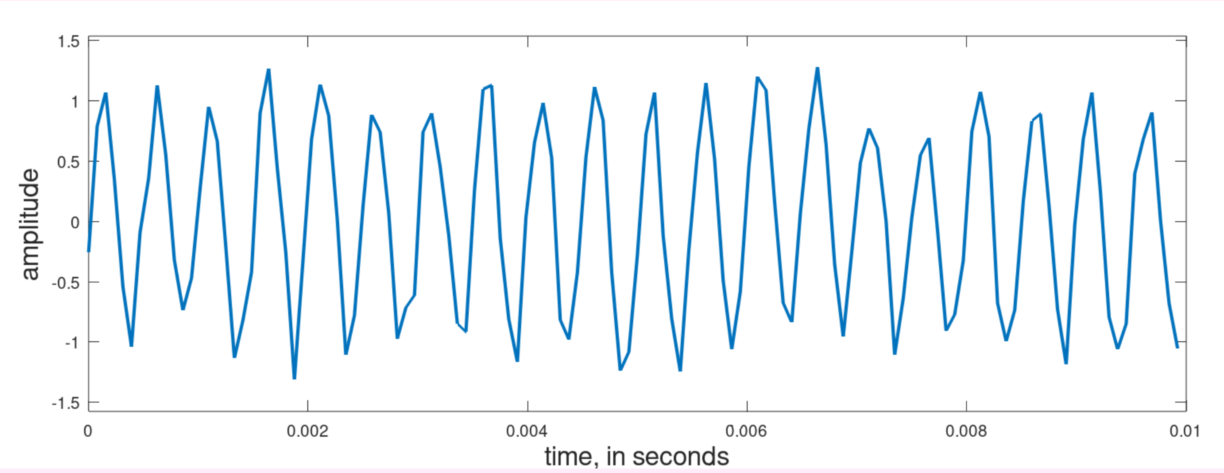

FIGURE 1.1: Time Domain Representation of Signal

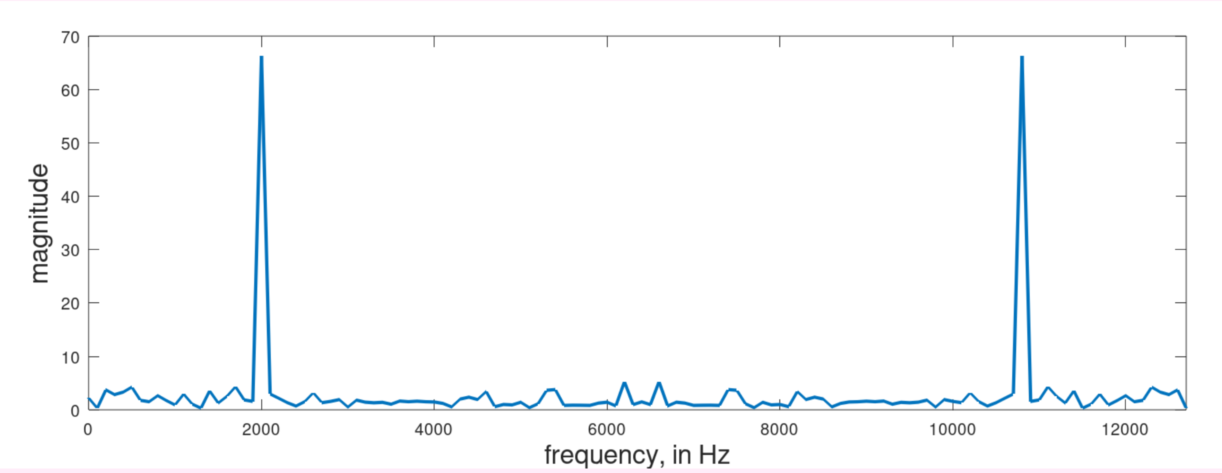

FIGURE 1.2: Magnitude of DFT of input-signal

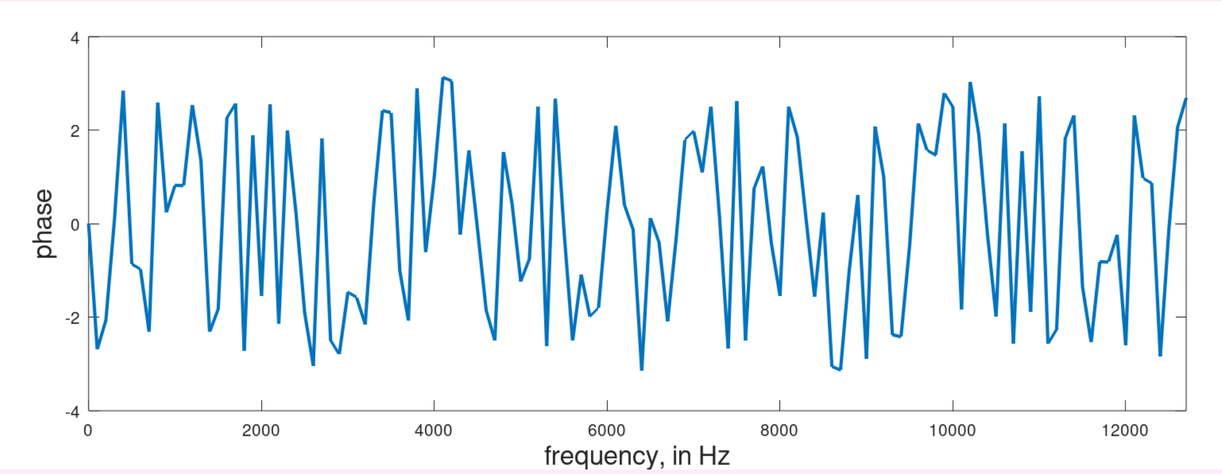

FIGURE 1.3: Phase of DFT of input-signal

1.3 C++ Code

```
// ==========================================================================
#include "include/before.hpp"
// main-file =================================================================
int main(){

    // starting timer
    auto logfile    {string("../csv-files/logfile.csv")};
    Timer timer(logfile);

    // init-variables
    auto f          {2000};                         // frequency of signal
    auto Fs         {12800};                        // sampling frequency
    auto Ts         {1/static_cast<double>(Fs)};    // corresponding time-period
    auto N          {128};                          // num-samples

```

```
    auto snr        {10};                                              // signal-to-noise ratio
    auto snrweight  {std::pow(10, (-1 * snr * 0.05))};                 // corresponding weight

    // building time-array
    vector<double> t(N,0); t.reserve(N);
    t = linspace(0.0, static_cast<double>(N-1), static_cast<size_t>(N)) * Ts;
    fWriteVector(t, "../csv-files/t-Objective1.csv");

    // creating sine-wave
    auto y = t;
    std::transform(t.begin(),
                   t.end(),
                   y.begin(),
                   [&](const auto x){return std::sin( 2 * std::numbers::pi * f * x);});

    // adding noise to the vector
    auto newmat     {y + snrweight * rand(-1.0, 1.0, y.size())};
    auto timeaxis   {linspace(static_cast<double>(0),
                              static_cast<double>((N-1)*Ts),
                              N)};
    fWriteVector(timeaxis,  "../csv-files/timeaxis-Objective1.csv");
    fWriteVector(newmat,    "../csv-files/newmat-Objective1.csv");

    // Taking the fourier transform
    auto nfft       {N};
    auto fend       {static_cast<double>((nfft-1) * Fs) / static_cast<double>(nfft)};
    auto waxis      {linspace(static_cast<double>(0),
                              static_cast<double>(fend),
                              nfft)};
    auto Fourier    {fft(newmat)};
    fWriteVector(waxis,     "../csv-files/waxis-Objective1.csv");
    fWriteVector(Fourier,   "../csv-files/Fourier-Objective1.csv");

    // return
    return(0);

}
```

1.4 Octave Code

```
%% Basic Setup
clc; clear all; close all;
addpath("./include/")

%% Loading the files
timeaxis    = csvread("../csv-files/timeaxis-Objective1.csv");
newmat      = csvread("../csv-files/newmat-Objective1.csv");
waxis       = csvread("../csv-files/waxis-Objective1.csv");
newmatfft   = fReadCSV("../csv-files/Fourier-Objective1.csv");
```

```

%% Plotting
plotwidth   = 1515;
plotheight  = 500;

figure(1)
set(gcf, 'Position', [0 0 plotwidth plotheight]);   % [left bottom width height]
plot(timeaxis, newmat, "LineWidth", 2);
xlabel("time, in seconds",  "fontsize", 16);
ylabel("amplitude",         "fontsize", 16);
ylim([1.2 * min(newmat), 1.2 * max(newmat)]);
saveas(gcf, "../Figures/y-objective1.png");

figure(2);
set(gcf, 'Position', [0 0 plotwidth plotheight]);   % [left bottom width height]
plot(waxis, abs(newmatfft), "LineWidth", 2);
xlabel("frequency, in Hz",  "fontsize", 16);
ylabel("magnitude",         "fontsize", 16);
xlim([min(waxis), max(waxis)]);
saveas(gcf, "../Figures/abs-yfft-objective1.png");

figure(3);
set(gcf, 'Position', [0 0 plotwidth plotheight]);   % [left bottom width height]
plot(waxis, angle(newmatfft), "LineWidth", 2);
xlabel("frequency, in Hz",  "fontsize", 16);
ylabel("phase",             "fontsize", 16);
xlim([min(waxis), max(waxis)]);
saveas(gcf, "../Figures/phase-yfft-objective1.png");
```

Chapter 2

Simulate the input to a 4 element array.

Here, the outputs of a 4 element array is simulated. The location of the source signal is far and therefore the wave fronts are parallel and arrives at the same angle at each element. As a result there would be a phase difference in the output of each element even for the same source signal due to difference in distance travelled. The sine wave is created as output from each element and corresponding phase difference is brought about. The noise is added by processing the SNR parameter.

2.1 Plots

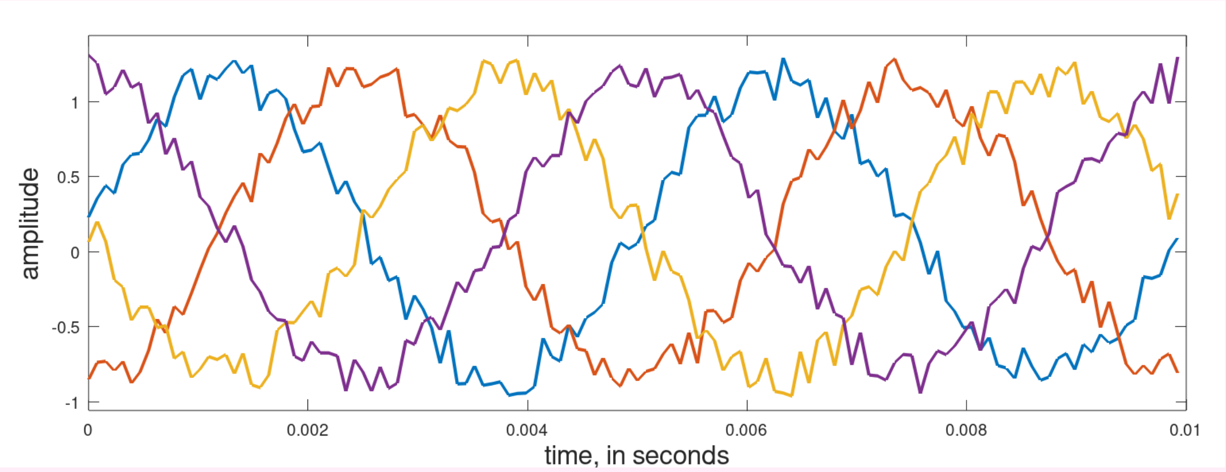

FIGURE 2.1: Time Domain Representation of Signal

2.2 C++ Code

```cpp
// ==========================================================================
#include "include/before.hpp"
// main-file ================================================================
int main(){

```

```
    // starting timer
    auto logfile    {string("../csv-files/logfile.csv")};
    Timer timer(logfile);

    // init-variables
    auto f                  {200};                                      // frequency of signal
    auto Fs                 {12800};                                    // sampling frequency
    auto Ts                 {1/static_cast<double>(Fs)};                // corresponding time-period
    auto N                  {128};                                      // num-samples

    auto m                  {4};
    auto angleofarrival     {60};
    auto speedofsound       {1500};
    auto lambda             {static_cast<double>(speedofsound)/static_cast<double>(f)};
    auto x                  {lambda/2};
    auto d                  {x * std::cos(angleofarrival * std::numbers::pi / 180) / speedofsound};

    auto snr                {10};                                       // signal-to-noise ratio
    auto snrweight          {std::pow(10, (-1 * snr * 0.05))};          // corresponding weight

    // building time-array
    vector<double> t(N,0); t.reserve(N);
    t = linspace(0.0,
                 static_cast<double>(N-1),
                 static_cast<size_t>(N)) * Ts;
    fWriteVector(t, "../csv-files/timeaxis-Objective2.csv");

    // building matrix
    auto matrix = Zeros({m, N});

    // creating sine-wave
    auto y = sin(2 * std::numbers::pi * f * t);
    fWriteVector(y, "../csv-files/y-Objective2.csv");

    // building the matrix
    for(int i = 0; i<m; ++i)
        matrix[i] = sin(2 * std::numbers::pi * f * (t  -  i * d ));
    fWriteMatrix(matrix, "../csv-files/matrix-Objective2.csv");

    // Adding noise to the matrix
    vector<vector<double>> additivenoise = snrweight * rand(0.0, 1.0, {m, N});
    fWriteMatrix(additivenoise, "../csv-files/additivenoise-Objective2.csv");

    auto newmat {matrix + additivenoise};
    fWriteMatrix(newmat, "../csv-files/newmat-Objective2.csv");

    // return
    return(0);

```

```
}
```

2.3 Octave Code

```
%% Basic Setup
clc; clear all; close all;
addpath("./include/")

%% Loading the files
timeaxis    = csvread("../csv-files/timeaxis-Objective2.csv");
newmat      = csvread("../csv-files/newmat-Objective2.csv");

%% Plotting
plotwidth   = 1515;
plotheight  = 500;

figure(1)
set(gcf, 'Position', [0 0 plotwidth plotheight]);   % [left bottom width height]
plot(timeaxis, newmat, "LineWidth", 2);
xlabel("time, in seconds",  "fontsize", 16);
ylabel("amplitude",         "fontsize", 16);
ylim([1.1 * min(newmat(:)), 1.1 * max(newmat(:))]);
saveas(gcf, "../Figures/newmat-Objective2.png");
```

Chapter 3

Narrowband beamformer

3.1 Aim

The outputs of each element in the array differ in phase. They are made in phase again by bringing about an artificial delay corresponding to the array position. In this code, this artificial delay is brought on the property of Fourier transform that

$$x(t - t_0) \Leftrightarrow e^{-j\omega t_0} X(\omega)$$

So the input is Fourier transformed. Then a weight vector is defined for each angle ranging from 0 to 180 degrees and the two matrices are multiplied. Then the absolute value vs angle is plotted.

3.2 Plots

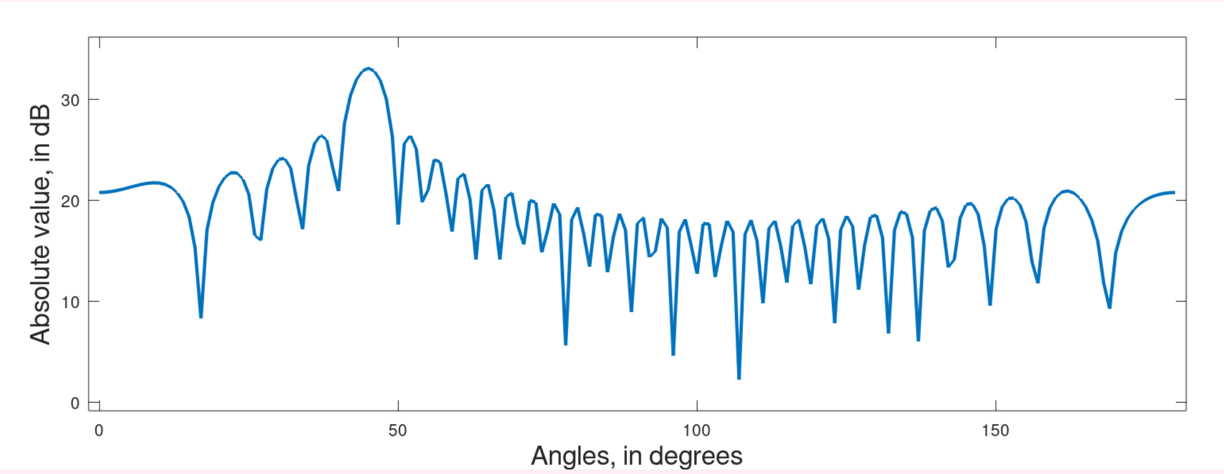

FIGURE 3.1: Beam-pattern for beamformer at angle 45

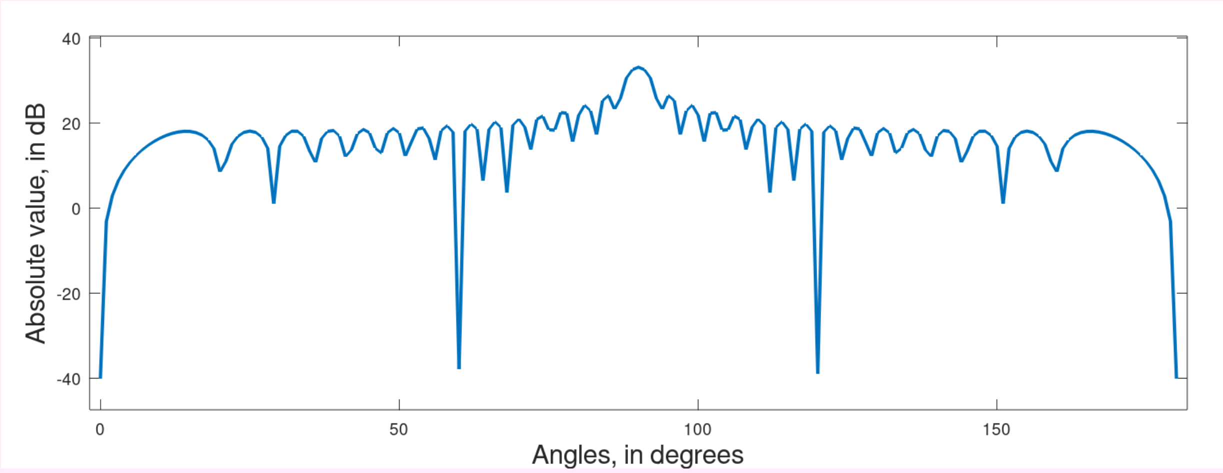

FIGURE 3.2: Beam-pattern for beamformer at angle 90

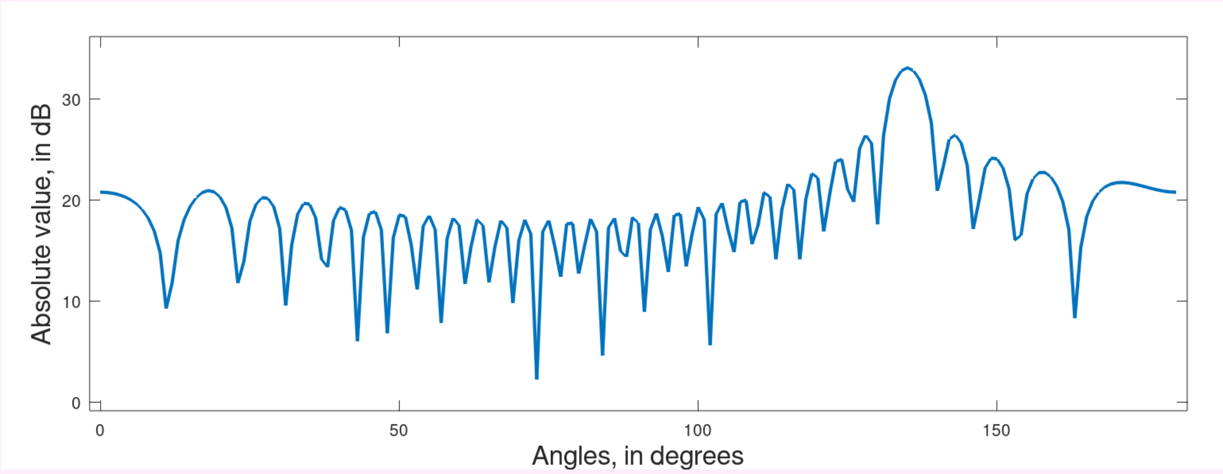

FIGURE 3.3: Beam-pattern for beamformer at angle 135

3.3 Observation

It is seen that for the angle equal to the source angle, a maxima is given and for every other angles, an absolute value much smaller than the peak value is given.

3.4 C++ Code

```
// ===============================================================================
#include "include/before.hpp"
// main-file =====================================================================
int main(){

    // starting timer
    auto logfile    {string("../csv-files/logfile-Objective3.csv")};
    Timer timer(logfile);

    // init-variables
```

```
    auto f                  {2000};                                      // frequency of signal
    auto Fs                 {12800};                                     // sampling frequency
    auto Ts                 {1.00/static_cast<double>(Fs)};              // corresponding time-period
    auto N                  {128};                                       // num-samples

    auto m                  {32};
    auto angleofarrival     {135};
    auto speedofsound       {1500};
    auto lambda {static_cast<double>(speedofsound)/static_cast<double>(f)};
    auto x                  {lambda/2.00};
    auto d                  {x * std::cos(static_cast<double>(angleofarrival) * std::numbers::pi / 180) / speedofsound};

    auto snr                {100};                                       // signal-to-noise ratio
    auto snrweight          {std::pow(10, (-1 * snr * 0.05))};           // corresponding weight


    // building time-array
    vector<double>t = linspace(0.0,
                               static_cast<double>(N-1) * Ts,
                               static_cast<size_t>(N));
    fWriteVector(t, "../csv-files/t-Objective3.csv");

    // building matrix
    auto matrix = Zeros({m, N});

    // creating sine-wave
    auto y                  {sin(2 * std::numbers::pi * f * t)};
    fWriteVector(y, "../csv-files/y-Objective3.csv");

    // building the matrix
    for(int sensorindex = 0; sensorindex < m; ++sensorindex)
        matrix[sensorindex] = sin(2.00 * std::numbers::pi * f * (t - sensorindex * d));
    fWriteMatrix(matrix, "../csv-files/matrix-Objective3.csv");


    // Adding noise to the matrix
    auto newmat             {matrix + snrweight * rand(0.0, 1.0, {m, N})};
    fWriteMatrix(newmat, "../csv-files/newmat-Objective3.csv");


    // Taking the fourier-transform
    auto nfft       {N};
    auto fend       {static_cast<double>(nfft - 1) * static_cast<double>(Fs) / static_cast<double>(nfft)};
    auto waxis      {linspace(0, fend, nfft)};
    auto Fourier    {fft(newmat, nfft)};
    fWriteMatrix(Fourier, "../csv-files/Fourier-Objective3.csv");

```

```

    // choosing the frequency row
    int index  = std::floor(static_cast<double>(f)/
    (static_cast<double>(Fs)/static_cast<double>(nfft)));
    auto fmat   {slice(Fourier, {-1, -2, index, index})};


    // Bringing the delay in frequency region
    auto anglematrix    {vector<double>(181)};
    auto delaycolumn    {vector<complex<double>>(m)};

    // moving through angle
    for(int testangle = 0; testangle<181; ++testangle){

        double testd  {x * cosd(testangle) / speedofsound};

        for(int currsensor = 0; currsensor < m; ++currsensor){
            delaycolumn[currsensor] = \
                std::exp( 1 * std::complex<double>{0, 1} * 2 * std::numbers::pi * f
    * currsensor * testd);
        }

        // calculating inner-product
        auto innerproduct_value {delaycolumn[0] * fmat[0][0]};
        for(int i = 1; i<delaycolumn.size(); ++i)
            innerproduct_value += delaycolumn[i] * fmat[i][0];

        // storing to the angle-matrix
        anglematrix[testangle] = std::abs(innerproduct_value);
    }
    fWriteVector(anglematrix, "../csv-files/anglematrix-Objective3.csv");

    // creating angle-axis
    auto angleaxis  {linspace(0, 180, 181)};
    fWriteVector(angleaxis, "../csv-files/angleaxis-Objective3.csv");

    // return
    return(0);

}
```

3.5 Octave Code

```
%% Basic Setup
clc; clear all;

%% Loading the files
timearray       = csvread("../csv-files/t-Objective3.csv");
yarray          = csvread("../csv-files/y-Objective3.csv");
matrix          = csvread("../csv-files/matrix-Objective3.csv");

ULAMatrix       = csvread("../csv-files/newmat-Objective3.csv");
FourierMatrix   = fReadCSV("../csv-files/Fourier-Objective3.csv");
```

```

angleaxis       = csvread("../csv-files/angleaxis-Objective3.csv");
anglematrix     = csvread("../csv-files/anglematrix-Objective3.csv");
anglematrixdB   = 10*log10(anglematrix);

%% Plotting the signals
plotwidth   = 1515;
plotheight  = 500;

figure(1);
set(gcf, 'Position', [0 0 plotwidth plotheight]);   % [left bottom width height]
plot(angleaxis, anglematrixdB, "LineWidth", 2);
xlabel("Angles, in degrees",         "fontsize", 16);
ylabel("Absolute value, in dB",      "fontsize", 16);
ylim([min(anglematrixdB) - 1e-1 * range(anglematrixdB),
      max(anglematrixdB)  + 1e-1 * range(anglematrixdB)]);
xlim([min(angleaxis) - 1e-2 * range(angleaxis),
      max(angleaxis) + 1e-2 * range(angleaxis)]);
saveas(gcf, "../Figures/anglematrixdB-Objective3.png");
```

Chapter 4

Simulate beam pattern by shifting theta

4.1 Aim

In this code, we explore the property of the beamformer in hand. The beam former is essentially a spatial filter. That is, it attenuates the signals which doesn't come from a specific direction (which we decide when designing) and provides a gain for signals coming in the direction. We show this by plotting the gain vs angle.

4.2 Plots

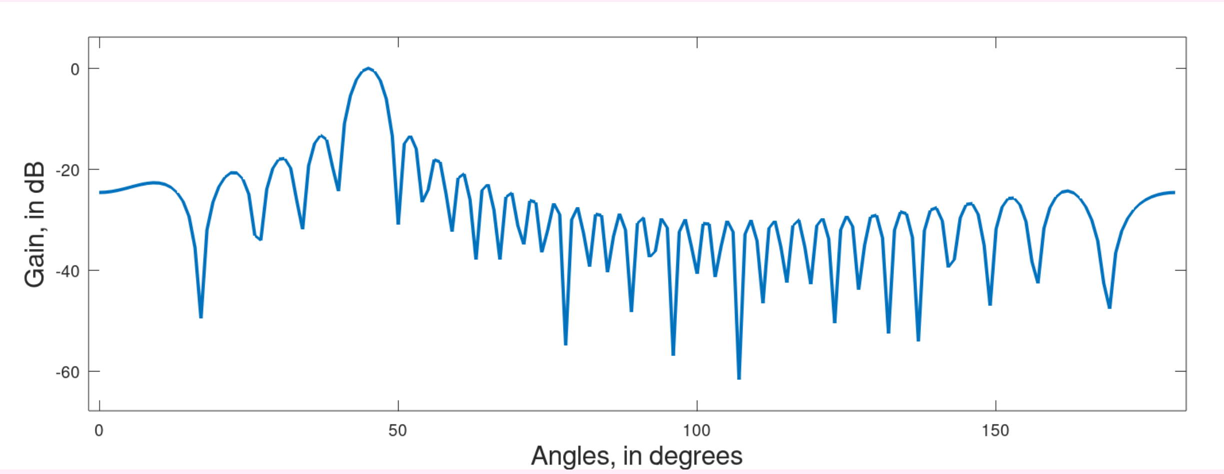

FIGURE 4.1: Beam-pattern for beamformer at angle 45

4.3 Observation

We can see that when we change the designed angle, the gain and attenuation correspondingly move to the designed angle.

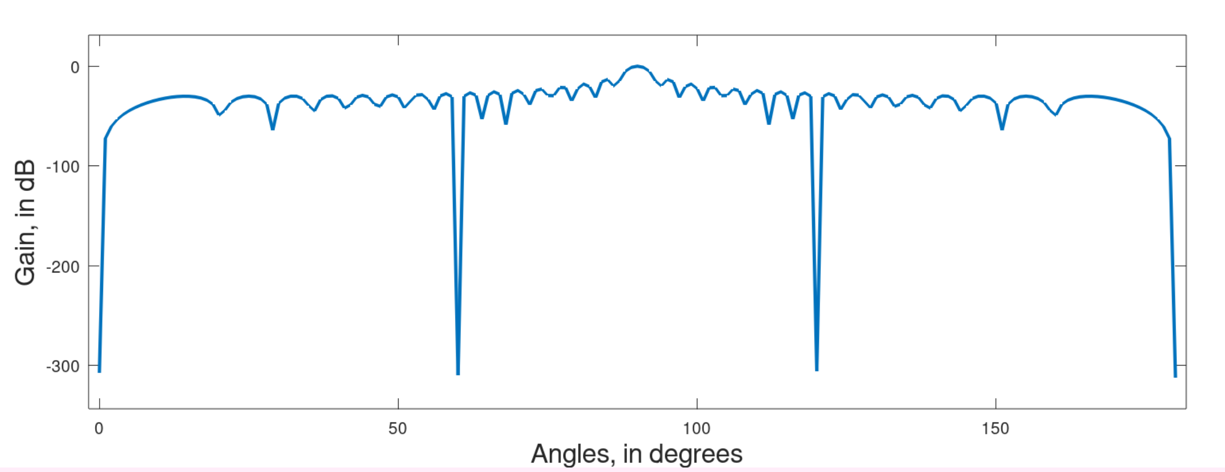

FIGURE 4.2: Beam-pattern for beamformer at angle 90

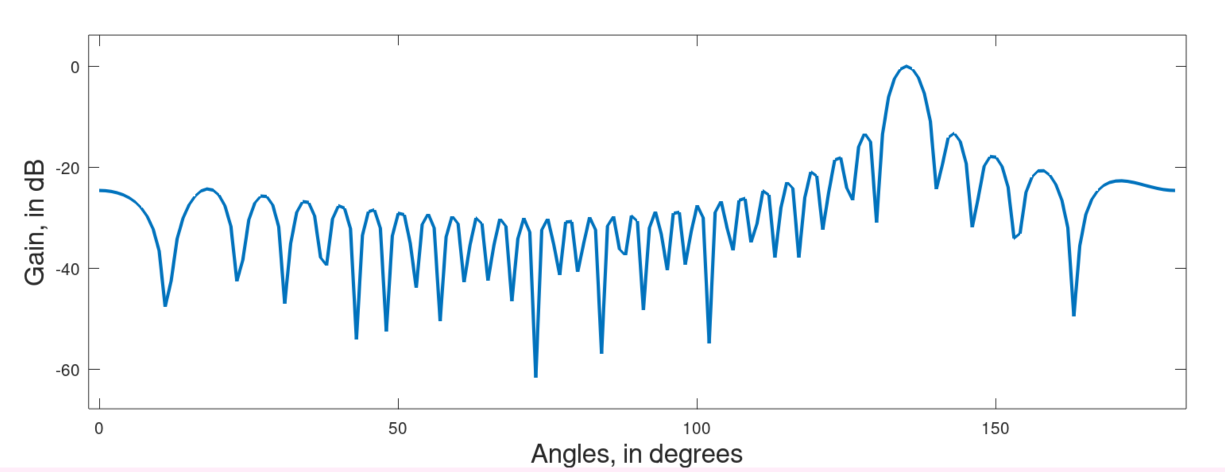

FIGURE 4.3: Beam-pattern for beamformer at angle 135

4.4 C++ Code

```cpp
// ==============================================================================
#include "include/before.hpp"
// main-file ====================================================================
int main(){

    // starting timer
    auto logfile    {string("logfile.csv")};
    Timer timer(logfile);

    // init-variables
    auto f                  {2000};
    auto m                  {static_cast<double>(32)};
    auto angle              {135};
    auto c                  {1500};
    auto lambda             {static_cast<double>(c)/static_cast<double>(f)};
    auto x                  {lambda/2};
    auto d                  {x * cosd(angle)/c};

```

```
    // bringing about the natural delay
    auto matrix             {vector<complex<double>>(m, complex<double>(0))};
    for(auto sensorindex = 0; sensorindex < m; ++sensorindex){
        matrix[sensorindex] = \
            (1.00/static_cast<double>(m)) * \
            std::exp(-1.00 * 1i * 2.00 * \
                     std::numbers::pi * f * (sensorindex) * d);
    }

    // bringing the delay in frequency region
    auto delaycolumn    = vector<complex<double>>(m, complex<double>(0));
    auto anglematrix    = vector<double>(181, 0);

    // calculating
    for(int testangle = 0; testangle < 181; ++testangle){

        auto testd  {x * cosd(testangle)/c};

        for(int sensorindex = 0; sensorindex < m; ++sensorindex)
            delaycolumn[sensorindex]  = \
                std::exp(1.00 * 1i * 2.00 * \
                         std::numbers::pi * f * sensorindex * testd);

        // performing inner-product
        anglematrix[testangle] = \
            std::abs(std::inner_product(matrix.begin(),
                                        matrix.end(),
                                        delaycolumn.begin(),
                                        complex<double>{0}));

    }

    // producing angle axis
    auto angleaxis  {linspace(0, 180, 181)};

    // saving the tensors
    fWriteVector(angleaxis,     "../Figures/angleaxis-Objective4.csv");
    fWriteVector(anglematrix,   "../Figures/anglematrix-Objective4.csv");

    // return
    return(0);

}
```

4.5 Octave Code

```
%% Basic Setup
clc; clear all; close all;

%% Loading the files
angleaxis           = csvread("../Figures/angleaxis-Objective4.csv");
anglematrix         = csvread("../Figures/anglematrix-Objective4.csv");
anglematrixinDB     = 20 * log10(anglematrix);
```

```
%% Plotting the signals
plotwidth   = 1515;
plotheight  = 500;

figure(1);
set(gcf, 'Position', [0 0 plotwidth plotheight]);   % [left bottom width height]
plot(angleaxis, anglematrixinDB, "LineWidth", 2);
xlabel("Angles, in degrees",  "fontsize", 16);
ylabel("Gain, in dB",         "fontsize", 16);
ylim([min(anglematrixinDB) - 1e-1 * range(anglematrixinDB),
      max(anglematrixinDB) + 1e-1 * range(anglematrixinDB)]);
xlim([min(angleaxis) - 1e-2 * range(angleaxis),
      max(angleaxis) + 1e-2 * range(angleaxis)]);
saveas(gcf, "../Figures/anglematrixinDB-Objective4.png");
```

Chapter 5

Beam Patterns for Frequencies Different from Design-Frequency

5.1 Aim

Designing an array for a frequency is done by placing the elements in an array at interspace distance of half the wavelength of the signal in scrutiny. With increasing interspace distance, there is an increasing probability of end fire anomaly. End fire anomaly is the phenomenon when beam pattern is plotted, another maxima other than the original maxima arises.

The array we designed is designed for the frequency of 2kHz. So, to see how change in frequency affects the beam pattern, what we have to do is change the weight vector to corresponding values of the frequency and plot absolute value vs angle .

5.2 Plots

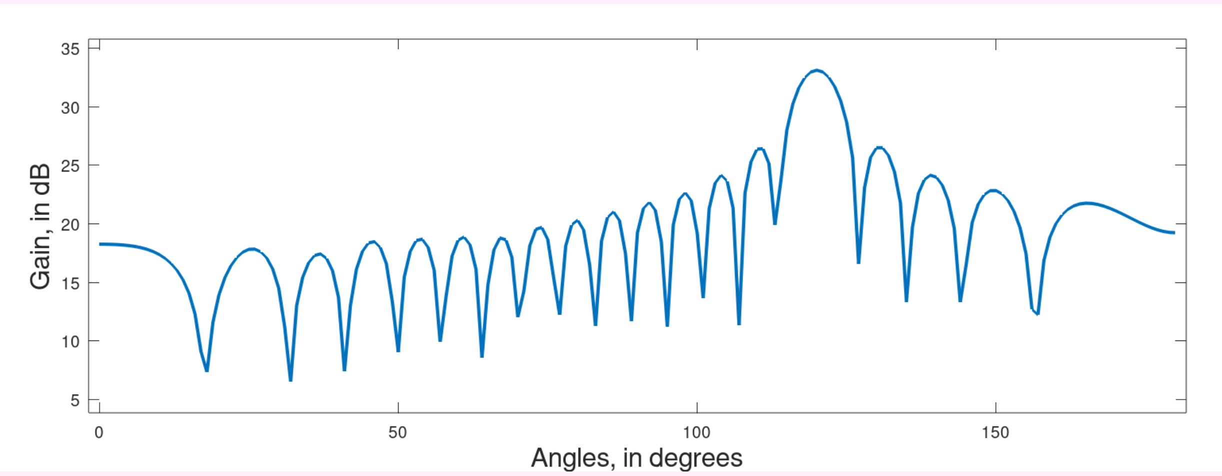

FIGURE 5.1: Beamformed for 1200Hz

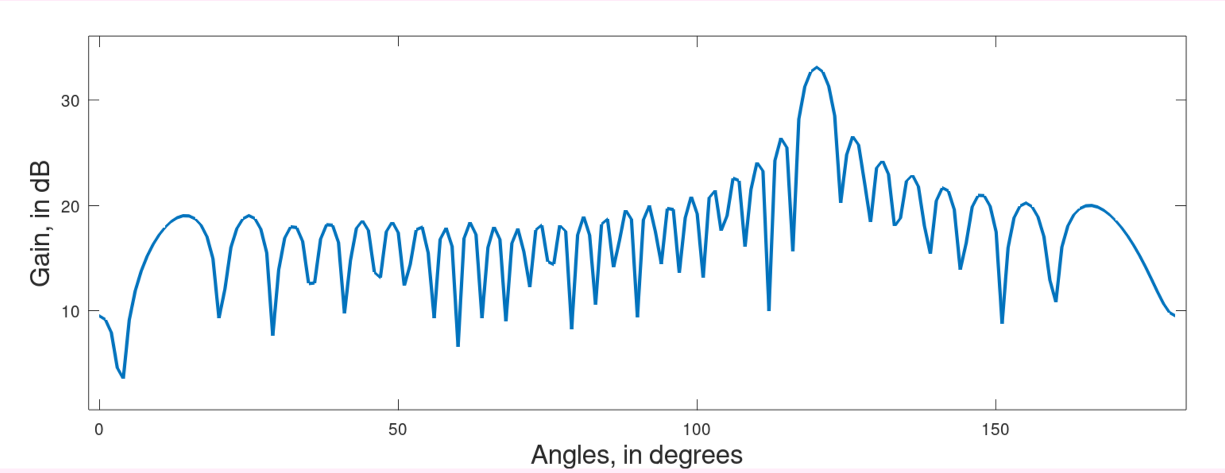

FIGURE 5.2: Beamformed for 2000Hz

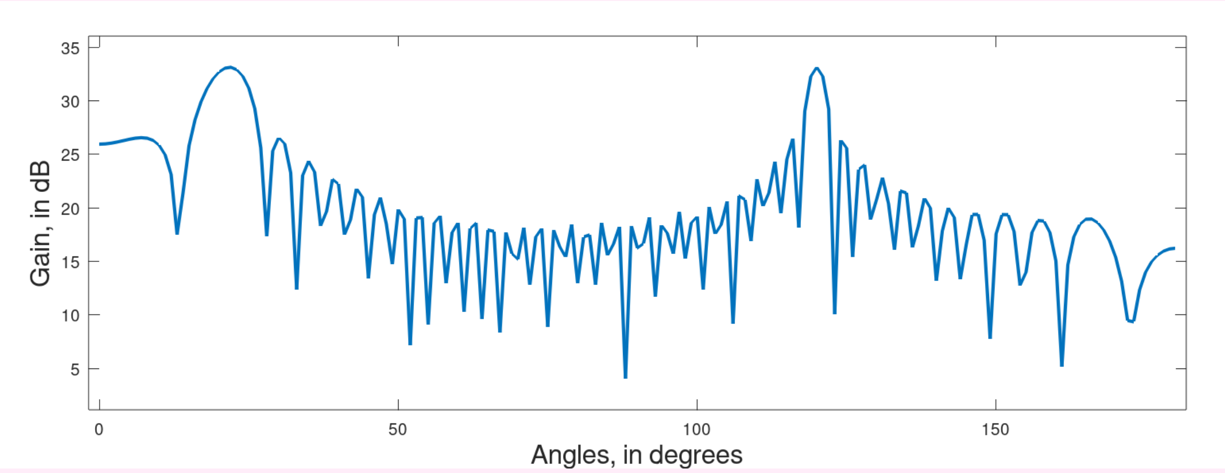

FIGURE 5.3: Beamformed for 2800Hz

5.3 Observation

We observe that as frequency goes below the value of the designed frequency, we still get a working beam pattern with no end fire anomaly. But as we increase the frequency beyond the designed frequency, another maxima arises. This poses the problem of ambiguity.

5.4 C++ Code

```
// ==============================================================================
#include "include/before.hpp"
// main-file ====================================================================
int main(){

    // starting timer
    auto logfile    {string("../csv-files/logfile.csv")};
```

```
    Timer timer(logfile);

    // init-variables
    auto f                  {2800};
    auto f_1                {2000};
    auto Fs                 {12800};
    auto Ts                 {1.00/static_cast<double>(Fs)};
    auto N                  {128};

    auto m                  {32};
    auto angle              {120};
    auto c                  {1500};
    auto lambda             {static_cast<double>(c)/static_cast<double>(f_1)};
    auto x                  {lambda/2.00};
    auto d                  {static_cast<double>(x * cosd(angle)/c)};

    auto snr                {10};
    auto snrweight          {static_cast<double>(std::pow(10, -1 * snr * 0.05))};


    // creating tensors
    auto t                  {linspace(static_cast<double>(0),
                                      static_cast<double>(N-1) * Ts,
                                      N)};
    auto matrix             {Zeros({m, N})};


    // bringing about natural delay
    for(int sensorindex = 0; sensorindex < m ; ++sensorindex)
        matrix[sensorindex] = \
            sin( 2.00 * std::numbers::pi * f * (t - sensorindex * d));
    fWriteVector(t,         "../csv-files/timearray-Objective5.csv");
    fWriteMatrix(matrix,    "../csv-files/matrix-Objective5.csv");


    // adding the noise
    auto newmat  = matrix + snrweight * rand(0.00, 1.00, {m, N});
    fWriteMatrix(newmat, "../csv-files/newmat-Objective5.csv");


    // taking the fourier transform
    auto nfft       {N};
    auto fend       {static_cast<double>((nfft-1) * Fs) /
    static_cast<double>(nfft)};
    auto waxis      {linspace(0, fend, nfft)};
    auto Fourier    {fft(newmat, nfft)};
    fWriteVector(waxis,     "../csv-files/waxis.csv");
    fWriteMatrix(Fourier,   "../csv-files/Fourier.csv");

    // choosing the frequency row
    int index       {static_cast<int>(std::floor(static_cast<double>(f)/
    (static_cast<double>(Fs)/static_cast<double>(nfft))))};
    auto fmat       {slice(Fourier, {-1, -2, index, index})};

    // bringing the delay in frequency region
    auto anglematrix    {vector<double>(181)};
    auto delaycolumn    {vector<complex<double>>(m, 0)};
```

```

    // building
    for(int testangle = 0; testangle < 181; ++testangle){

        auto testd  {x * cosd(testangle)/c};

        for(int sensorindex = 0; sensorindex < m; ++sensorindex)
            delaycolumn[sensorindex]    = std::exp(1 * 1i * 2 * std::numbers::pi *
    f * sensorindex * testd);

        anglematrix[testangle] = \
            std::abs(std::inner_product(fmat.begin(), fmat.end(),
                                        delaycolumn.begin(),
                                        complex<double>{0},
                                        std::plus<complex<double>>(),
                                        [](vector<complex<double>> a,
    complex<double> b){
                                            return a[0]*b;
                                        }));
    }

    // building axes
    auto angleaxis   = linspace(0, 180, 181);

    // saving
    fWriteVector(angleaxis,     "../csv-files/angleaxis-Objective-5.csv");
    fWriteVector(anglematrix,   "../csv-files/anglematrix-Objective-5.csv");

    // return
    return(0);

}
```

5.5 Octave Code

```
%% Basic Setup
clc; clear all;

%% Loading the files
angleaxis       = csvread("../csv-files/angleaxis-Objective-5.csv");
anglegains      = csvread("../csv-files/anglematrix-Objective-5.csv");
anglegainsindB  = 10 * log10(anglegains);

%% Plotting the signals
plotwidth    = 1515;
plotheight   = 500;

figure(1);
set(gcf, 'Position', [0 0 plotwidth plotheight]);   % [left bottom width height]
plot(angleaxis, anglegainsindB, "LineWidth", 2);
xlabel("Angles, in degrees",  "fontsize", 16);
ylabel("Gain, in dB",         "fontsize", 16);
```

```
ylim([min(anglegainsindB) - 1e-1 * range(anglegainsindB), max(anglegainsindB) +
    1e-1 * range(anglegainsindB)]);
xlim([min(angleaxis) - 1e-2 * range(angleaxis), max(angleaxis) + 1e-2 *
    range(angleaxis)]);
saveas(gcf, "../Figures/anglegainsindB-Objective5.png");
```

Chapter 6

Effect of SNR on beam pattern

6.1 Aim

SNR or signal to noise ratio plays an important role in beamforming. The beamformer is a spatial filter. That is, it gives a gain for signal coming from a certain angle/direction. This is an added advantage because along with attenuating other signals, it enables us to still find the source location even if the SNR is low.

In this code, we intend to see how change in SNR affects the relative side lobe levels. And to what extent we can find the source angle without ambiguity arising. The array is broadside beamformed.

6.2 Plots

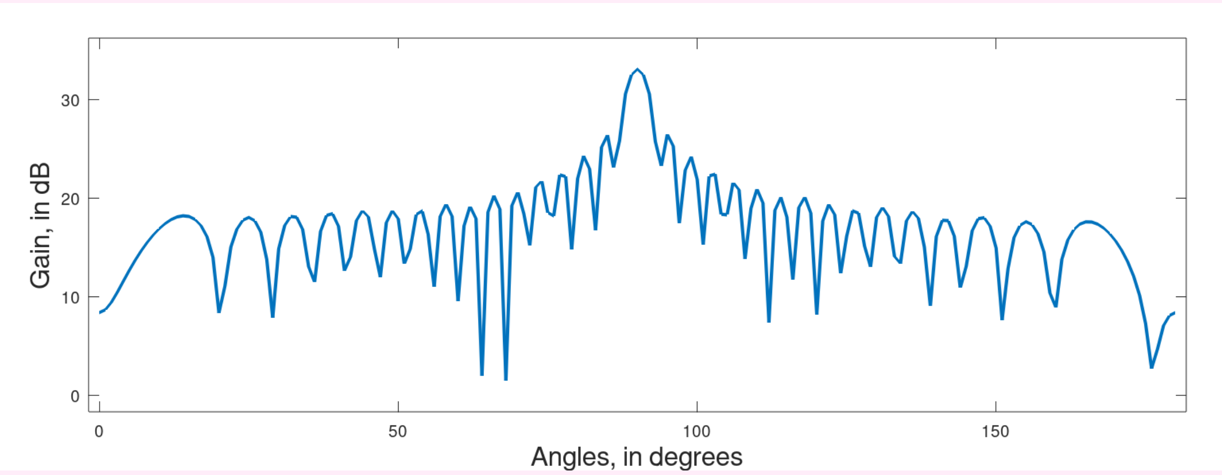

FIGURE 6.1: Beamforming for SNR = 10

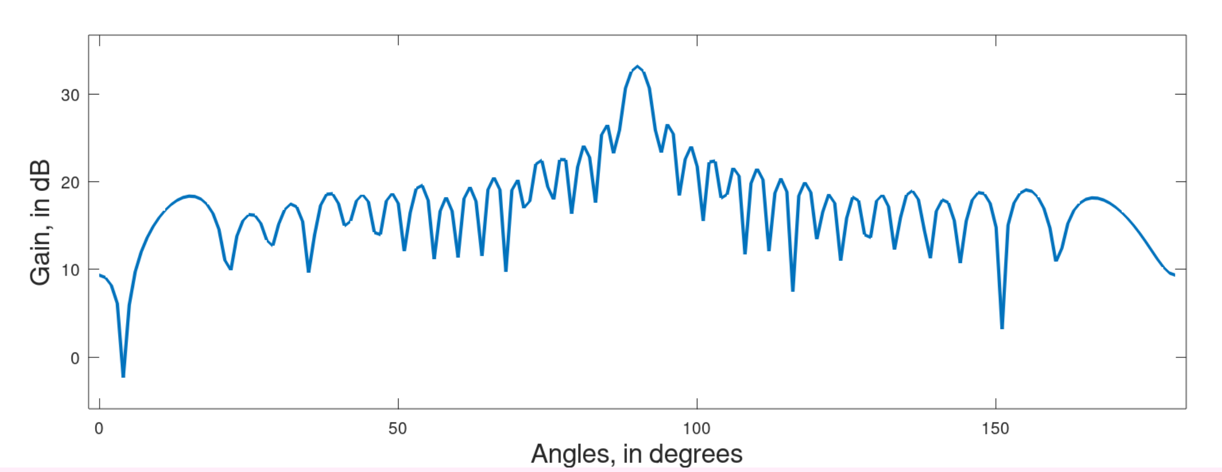

FIGURE 6.2: Beamforming for SNR = -1

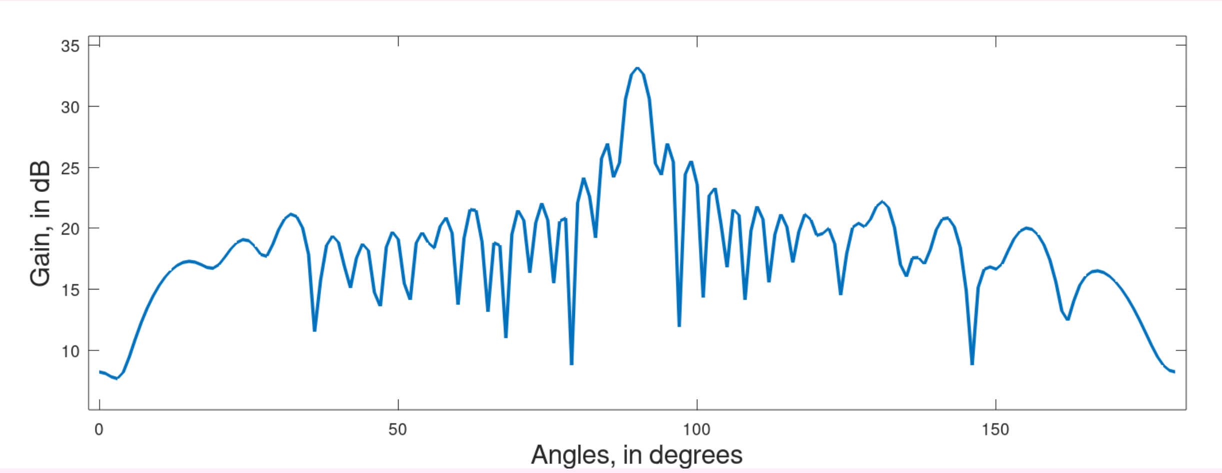

FIGURE 6.3: Beamforming for SNR = -10

6.3 Observation

We see that as SNR decreases, the relative side lobe levels rise and eventually will reach a level where we cannot differentiate between the side lobe and the main lobe.Smaller the value of SNR, more difficult it is to determine the source signal location.

6.4 C++ Code

```
// ==============================================================================
#include "include/before.hpp"
// main-file ===================================================================
int main(){

    // starting timer
    auto logfile    {string("../csv-files/logfile-Objective6.csv")};
    Timer timer(logfile);
```

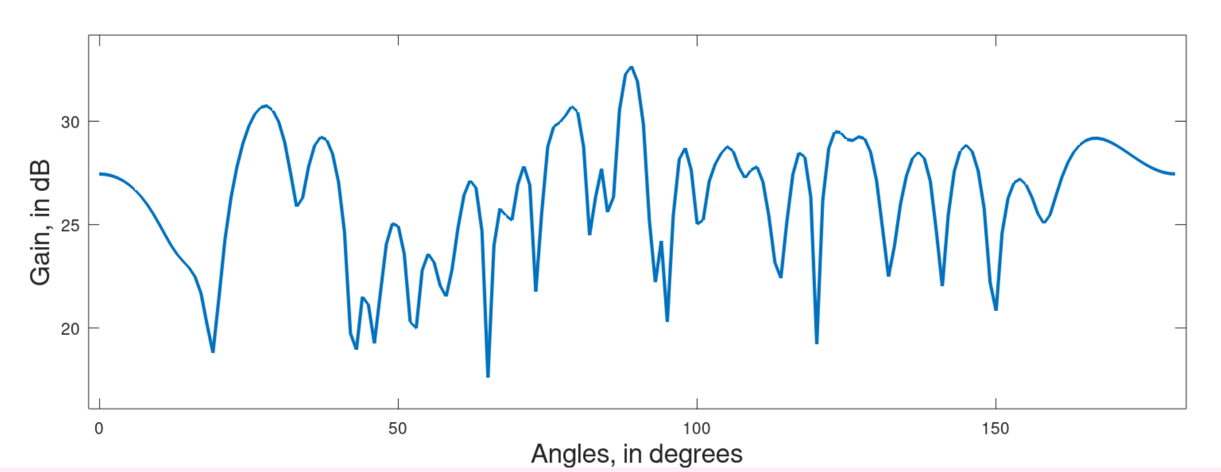

FIGURE 6.4: Beamforming for SNR = -30

```cpp

    // init-variables
    auto f                      {2800};
    auto Fs                     {12800};
    auto Ts                     {1.00/static_cast<double>(Fs)};
    auto N                      {128};

    auto m                      {32};
    auto angle                  {90};
    auto c                      {1500};
    auto lambda                 {static_cast<double>(c)/static_cast<double>(f)};
    auto x                      {lambda/2.00};
    auto d                      {static_cast<double>(x * cosd(angle)/c)};

    auto snr                    {30};
    auto snrweight              {static_cast<double>(std::pow(10, -1 * snr * 0.05))};


    // simulating signals
    auto t                      {linspace(static_cast<double>(0),
                                          static_cast<double>(N-1) * Ts,
                                          N)};
    auto matrix                 {Zeros({m, N})};


    // bringing about natural delay
    for(int sensorindex = 0; sensorindex < m ; ++sensorindex)
        matrix[sensorindex] = \
            sin( 2.00 * std::numbers::pi * f * (t - sensorindex * d));
    fWriteVector(t,             "../csv-files/timearray-Objective6.csv");
    fWriteMatrix(matrix,        "../csv-files/matrix-Objective6.csv");


    // adding the noise
    auto newmat  = matrix + snrweight * rand(0.00, 1.00, {m, N});
    fWriteMatrix(newmat, "../csv-files/newmat-Objective6.csv");


    // taking the fourier transform
```

```
    auto nfft       {N};
    auto fend       {static_cast<double>((nfft-1) * Fs) /
    static_cast<double>(nfft)};
    auto waxis      {linspace(0, fend, nfft)};
    auto Fourier    {fft(newmat, nfft)};
    fWriteVector(waxis,     "../csv-files/waxis.csv");
    fWriteMatrix(Fourier,   "../csv-files/Fourier.csv");

    // choosing the frequency row
    int index       {static_cast<int>(std::floor(static_cast<double>(f)/
    (static_cast<double>(Fs)/static_cast<double>(nfft))))};
    auto fmat       {slice(Fourier, {-1, -2, index, index})};

    // bringing the delay in frequency region
    auto anglematrix    {vector<double>(181)};
    auto delaycolumn    {vector<complex<double>>(m, 0)};

    // building
    for(int testangle = 0; testangle < 181; ++testangle){

        auto testd  {x * cosd(testangle)/c};

        for(int sensorindex = 0; sensorindex < m; ++sensorindex)
            delaycolumn[sensorindex]    = std::exp(1 * 1i * 2 * std::numbers::pi *
    f * sensorindex * testd);

        anglematrix[testangle] = \
            std::abs(std::inner_product(fmat.begin(), fmat.end(),
                                        delaycolumn.begin(),
                                        complex<double>{0},
                                        std::plus<complex<double>>(),
                                        [](vector<complex<double>> a,
    complex<double> b){
                                            return a[0]*b;
                                        }));
    }

    // building axes
    auto angleaxis   = linspace(0, 180, 181);

    // saving
    fWriteVector(angleaxis,     "../csv-files/angleaxis-Objective-6.csv");
    fWriteVector(anglematrix,   "../csv-files/anglematrix-Objective-6.csv");

    // return
    return(0);

}
```

6.5 Octave Code

```
%% Basic Setup
clc; clear all;
```

```
%% Loading the files
angleaxis       = csvread("../csv-files/angleaxis-Objective-6.csv");
anglegains      = csvread("../csv-files/anglematrix-Objective-6.csv");
anglegainsindB  = 10 * log10(anglegains);

%% Plotting the signals
plotwidth   = 1515;
plotheight  = 500;

figure(1);
set(gcf, 'Position', [0 0 plotwidth plotheight]);   % [left bottom width height]
plot(angleaxis, anglegainsindB, "LineWidth", 2);
xlabel("Angles, in degrees",  "fontsize", 16);
ylabel("Gain, in dB",         "fontsize", 16);
ylim([min(anglegainsindB) - 1e-1 * range(anglegainsindB), max(anglegainsindB) +
    1e-1 * range(anglegainsindB)]);
xlim([min(angleaxis) - 1e-2 * range(angleaxis), max(angleaxis) + 1e-2 *
    range(angleaxis)]);
saveas(gcf, "../Figures/anglegainsindB-Objective6.png");
```

Chapter 7

Reconstructing Original Signal from Beamformed Output

7.1 Aim:

Once the element signal outputs are fourier transformed and beamformed, it is essential to bring back the signal to the time domain. Since we deal with discrete series here, a problem arises because of cyclic delay.The last few samples would be error filled. We overcome this problem by using overlap-and-save method.

Here, two blocks are taken from the input matrix and then fourier transform is taken separately. And after beamforming, we take the inverse fourier transform. After taking the inverse fourier of the two blocks, we concatenate each other after removing the last few parts of both the blocks.

7.2 Observation

Figure 7.1 is the first block. We take a fixed number of samples and treat it like a new signal and this is called a block. Similarly Figure 7.2 shows the second block. Figure 7.3 is the final result. The one we get after concatenating the two blocks after removing a few samples from both of them. We can see that this is a good approximation of the initial signal.

Appendix A

C++ Function Definitions

A.1 before.hpp

```cpp
// including header-files
#include <algorithm>
#include <complex>
#include <bitset>
#include <climits>
#include <cstddef>
#include <iostream>
#include <limits>
#include <map>
#include <new>
#include <stdlib.h>
#include <unordered_map>
#include <vector>
#include <set>
#include <numeric>
#include <fstream>
#include <numbers>
#include <cmath>
#include <random>

// custom definitions
#include "hashdefines.hpp"
#include "usings.hpp"
#include "DataStructureDefinitions.hpp"
#include "PrintContainers.hpp"
#include "TimerClass.hpp"
#include "utils.hpp"
```

A.2 utils.hpp

```cpp
// ===========================================================================
#include "svr_WriteToCSV.hpp"
// ===========================================================================
template <typename F, typename R>
```

```
constexpr auto fElementWise(F&& func, R& range){
    std::transform(std::begin(range),
                   std::end(range),
                   std::begin(range),
                   std::forward<F>(func));
    // return range;
}
// ===========================================================================
#include "svr_repmat.hpp"
// ===========================================================================
auto SineElementWise(auto& input, auto constantvalue){
    for(auto& x: input) {x = std::sin(constantvalue * x);}
    // replace this with std::transform
};
// ===========================================================================
#include "svr_linspace.hpp"
// ===========================================================================
#include "svr_fft.hpp"
// ===========================================================================
template <typename T>
auto abs(vector<complex<T>> inputvector){
    vector<T> temp(inputvector.size(), 0);
    std::transform(temp.begin(),
                   temp.end(),
                   temp.begin(),
                   [](T a){return std::abs(a);});
    return temp;
}
// ===========================================================================
#include "svr_rand.hpp"
// ===========================================================================
#include "svr_operator_star.hpp"
// ===========================================================================
#include "svr_operators.hpp"
// ===========================================================================
#include "svr_tensor_inits.hpp"
// ===========================================================================
#include "svr_sin.hpp"
// ===========================================================================
#include "svr_slice.hpp"
// ===========================================================================
#include "svr_matrix_operations.hpp"
// ===========================================================================
#include <boost/type_index.hpp>
template <typename T>
auto type(T inputarg){
    std::cout <<
    boost::typeindex::type_id_with_cvr<decltype(inputarg)>().pretty_name()<< "\n";
}
// ===========================================================================
#include "svr_shape.hpp"
// ===========================================================================
// ===========================================================================
// ===========================================================================
// ===========================================================================
// ===========================================================================
// ===========================================================================
```

```
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
// ==============================================================================
```

A.3 hashdefines.hpp

```
// hash-deinfes
#define PRINTSPACE  std::cout << "\n\n\n\n" << std::endl;
#define PRINTLINE   std::cout << "===============================================" << std::endl;
```

A.4 usings.hpp

```
// borrowing from namespace std
using std::cout;
using std::complex;
using std::endl;
using std::vector;
using std::string;
using std::unordered_map;
using std::map;
using std::format;
using std::deque;
using std::pair;
using std::min;
using std::max;
using namespace std::complex_literals;
```

A.5 DataStructureDefinitions.hpp

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
    right(right) {}
};


struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};
```

A.6 PrintContainers.hpp

```
// vector printing function
template<typename T>
void fPrintVector(vector<T> input){
    for(auto x: input) cout << x << ",";
    cout << endl;
}

template<typename T>
void fpv(vector<T> input){
    for(auto x: input) cout << x << ",";
    cout << endl;
}

template<typename T>
void fPrintMatrix(vector<T> input){
    for(auto x: input){
        for(auto y: x){
            cout << y << ",";
        }
        cout << endl;
    }
}

template<typename T, typename T1>
void fPrintHashmap(unordered_map<T, T1> input){
    for(auto x: input){
        cout << format("[{},{}] | ", x.first, x.second);
    }
    cout <<endl;
```

```
}

void fPrintBinaryTree(TreeNode* root){
    // sending it back
    if (root == nullptr) return;

    // printing
    PRINTLINE
    cout << "root->val = " << root->val << endl;

    // calling the children
    fPrintBinaryTree(root->left);
    fPrintBinaryTree(root->right);

    // returning
    return;

}

void fPrintLinkedList(ListNode* root){
    if (root == nullptr) return;
    cout << root->val << " -> ";
    fPrintLinkedList(root->next);
    return;
}

template<typename T>
void fPrintContainer(T input){
    for(auto x: input) cout << x << ", ";
    cout << endl;
    return;
}
// ============================================================================
template <typename T>
auto size(std::vector<std::vector<T>> inputMatrix){
    cout << format("[{}, {}]\n", inputMatrix.size(), inputMatrix[0].size());
}

template <typename T>
auto size(const std::string inputstring, std::vector<std::vector<T>> inputMatrix){
    cout << format("{} = [{}, {}]\n", inputstring, inputMatrix.size(),
    inputMatrix[0].size());
}
```

A.7 TimerClass.hpp

```
struct Timer
{
    std::chrono::time_point<std::chrono::high_resolution_clock> startpoint;
    std::chrono::time_point<std::chrono::high_resolution_clock> endpoint;
    std::chrono::duration<long long, std::nano>                 duration;
    std::string                                                 filename;
    std::string                                                 functionname;
```

```cpp

    // constructor
    Timer()         {start();}
    Timer(std::string logfile_arg): filename(std::move(logfile_arg)) {start();}
    Timer(std::string logfile_arg,
          std::string func_arg): filename(std::move(logfile_arg)),
                                 functionname(std::move(func_arg))  {start();}

    void start()    {startpoint = std::chrono::high_resolution_clock::now();}
    void stop()     {endpoint   = std::chrono::high_resolution_clock::now(); fetchtime();}

    void fetchtime(){
        duration = std::chrono::duration_cast<std::chrono::nanoseconds>(endpoint - startpoint);
        cout << format("{}  nanoseconds \n", duration.count());
    }
    void fetchtime(string stringarg){
        duration = std::chrono::duration_cast<std::chrono::nanoseconds>(endpoint - startpoint);
        cout << format("{} took {}  nanoseconds \n", stringarg, duration.count());
    }
    void measure(){
        auto temp = std::chrono::high_resolution_clock::now();
        auto nsduration  = std::chrono::duration_cast<std::chrono::nanoseconds>(temp - startpoint);
        auto msduration  = std::chrono::duration_cast<std::chrono::microseconds>(temp - startpoint);
        auto sduration   = std::chrono::duration_cast<std::chrono::seconds>(temp - startpoint);
        cout << format("{} nanoseconds | {} microseconds | {} seconds \n",
            nsduration.count(), msduration.count(), sduration.count());
    }
    ~Timer(){
        auto temp = std::chrono::high_resolution_clock::now();
        auto nsduration  = std::chrono::duration_cast<std::chrono::nanoseconds>(temp - startpoint);
        auto msduration  = std::chrono::duration_cast<std::chrono::microseconds>(temp - startpoint);
        auto milliduration = std::chrono::duration_cast<std::chrono::milliseconds>(temp - startpoint);
        auto sduration   = std::chrono::duration_cast<std::chrono::seconds>(temp - startpoint);
        PRINTLINE
        cout << format("{} nanoseconds | {} microseconds | {} milliseconds | {} seconds \n",
            nsduration.count(), msduration.count(), milliduration.count(), sduration.count());

        // writing to the file
        if (!filename.empty()){
            std::ofstream fileobj(filename, std::ios::app);
            if (fileobj){
                if (functionname.empty()){
                    fileobj << "main" << "," << nsduration.count() << "," << msduration.count() << "," << sduration.count() << "\n";
                }
```

```
                else{
                    fileobj << functionname << "," << nsduration.count() << "," <<
    msduration.count() << "," << sduration.count() << "\n";
                }
            }
        }
    }
};
```

Appendix B

Octave Function Definitions

B.1 fReadCSV.m

```
function finalmatrix = fReadCSV(filename_string)
    entire_text             = fileread(filename_string);
    total_num_characters    = numel(entire_text);
    p1                      = 1;
    p2                      = 1;
    currentry               = "";
    currline                = [];
    finalmatrix             = [];

    while(p2 <= total_num_characters)

        curr_char = entire_text(p2);

        if(curr_char == ',' || curr_char == "\n")
            curr_entry  = entire_text(p1:p2-1);
            currline    = [currline, str2num(curr_entry)];

            if (curr_char == "\n")
                finalmatrix = [finalmatrix; currline];
                currline    = [];
            end

            p1 = p2 + 1;
        end
        p2 += 1;
    end
end
```