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 seafloor 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 ($\mathrm{J\,s^{-1}}$)
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

In this simulation, a sinusoidal signal is generated to emulate the conditions relevant to beamforming and to examine the response of a single array element. Noise is incorporated according to the specified signal-to-noise ratio (SNR) parameter, and the effects of varying SNR are subsequently analyzed. The signal behavior is observed in both the time and frequency domains: in the time domain, changes are examined by plotting amplitude as a function of time, while in the frequency domain, the signal is Fourier transformed and the magnitude spectrum is plotted as a function of frequency.

1.2 Plots

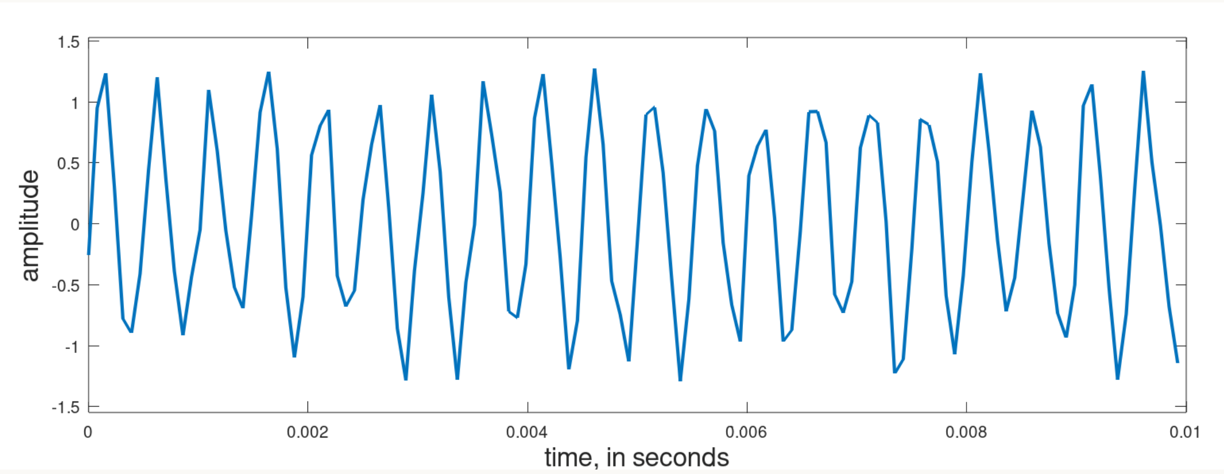

FIGURE 1.1: Time Domain Representation of Signal

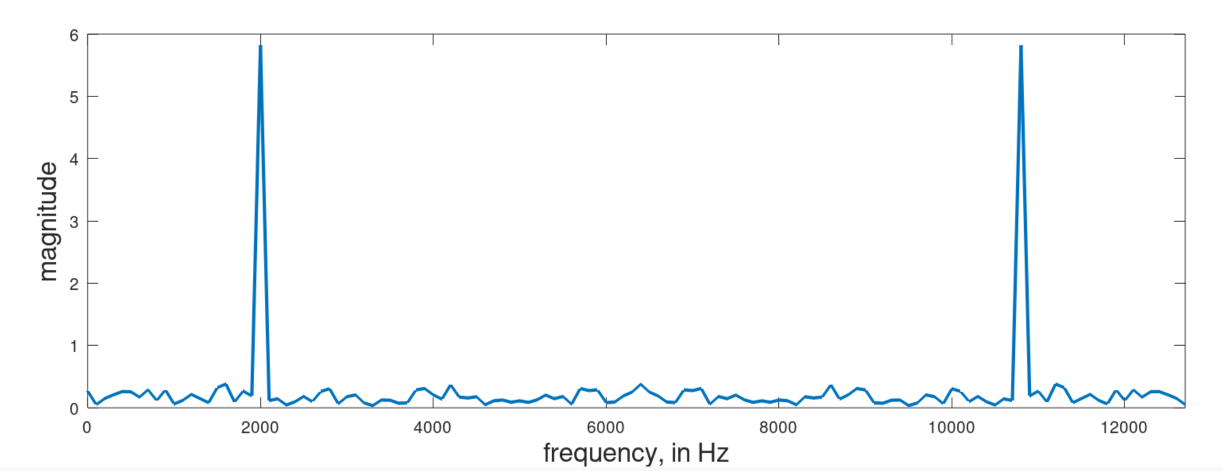

FIGURE 1.2: Magnitude of DFT of input-signal

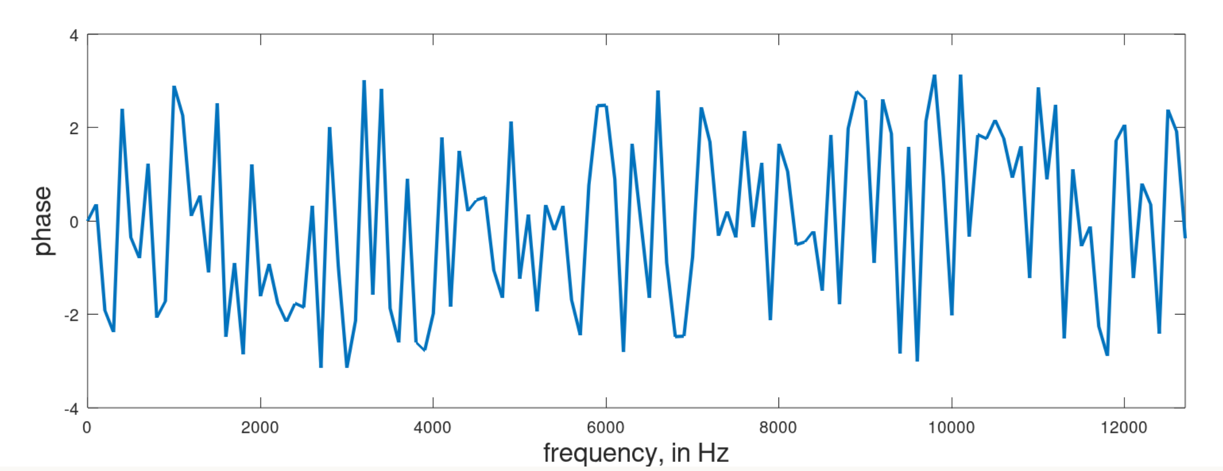

FIGURE 1.3: Phase of DFT of input-signal

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.

2.1 Aim

In this simulation, the outputs of a four-element array are modeled. The source is assumed to be located in the far field, such that the incident wavefronts are approximately planar and impinge on all array elements at the same angle. Consequently, despite originating from the same source, the signals received at different elements exhibit phase differences due to variations in propagation distance. The outputs of the elements are generated as sinusoids with the appropriate phase offsets, and noise is incorporated according to the specified signal-to-noise ratio (SNR) parameter.

2.2 Plots

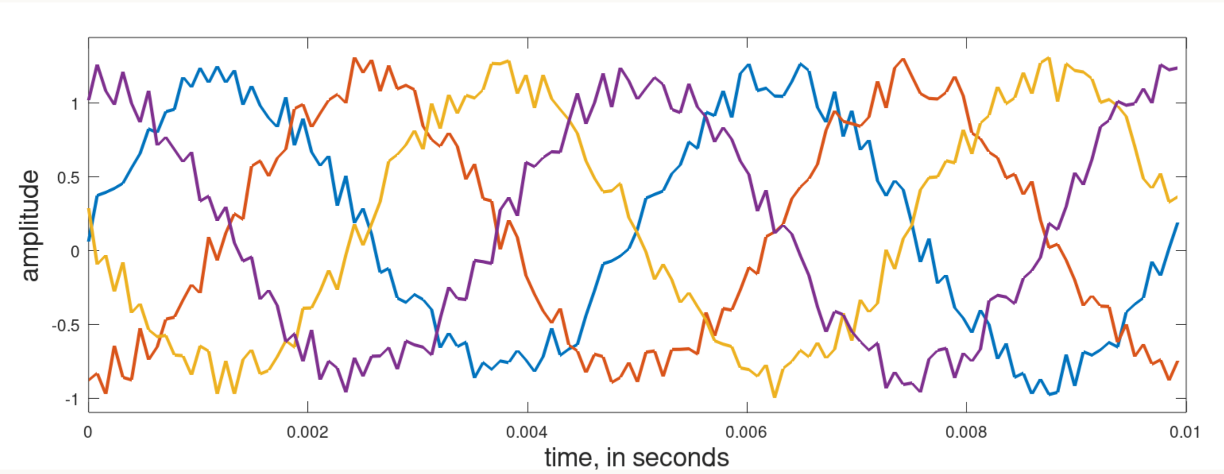

FIGURE 2.1: Time Domain Representation of Signal

2.3 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);

}
```

2.4 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 the individual array elements exhibit phase differences due to their spatial positions. Phase alignment is achieved by introducing an artificial delay that compensates for the relative displacement of each element. In this implementation, the delay is realized by exploiting the Fourier transform property

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

Accordingly, the input signal is first Fourier transformed. A weight vector is then defined for each steering angle in the range of 0 to 180 degrees, and subsequently multiplied with the transformed signal. Finally, the magnitude response is plotted as a function of angle.

3.2 Plots

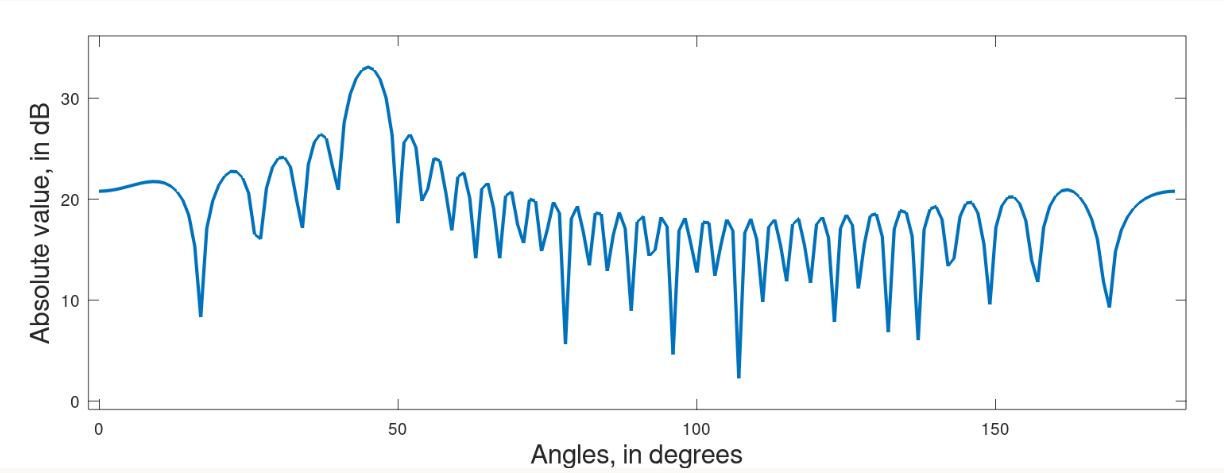

FIGURE 3.1: Beam-pattern for beamformer at angle 45

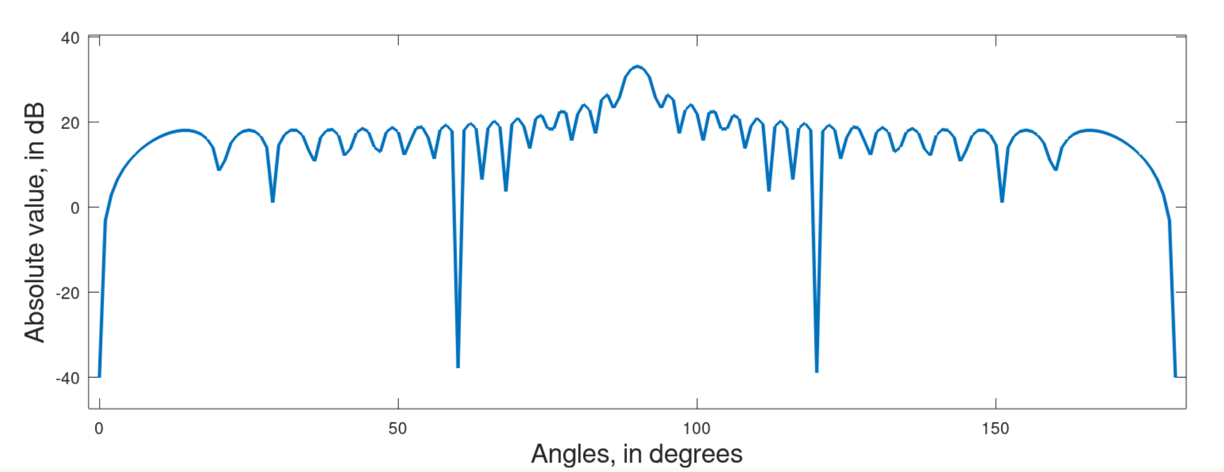

FIGURE 3.2: Beam-pattern for beamformer at angle 90

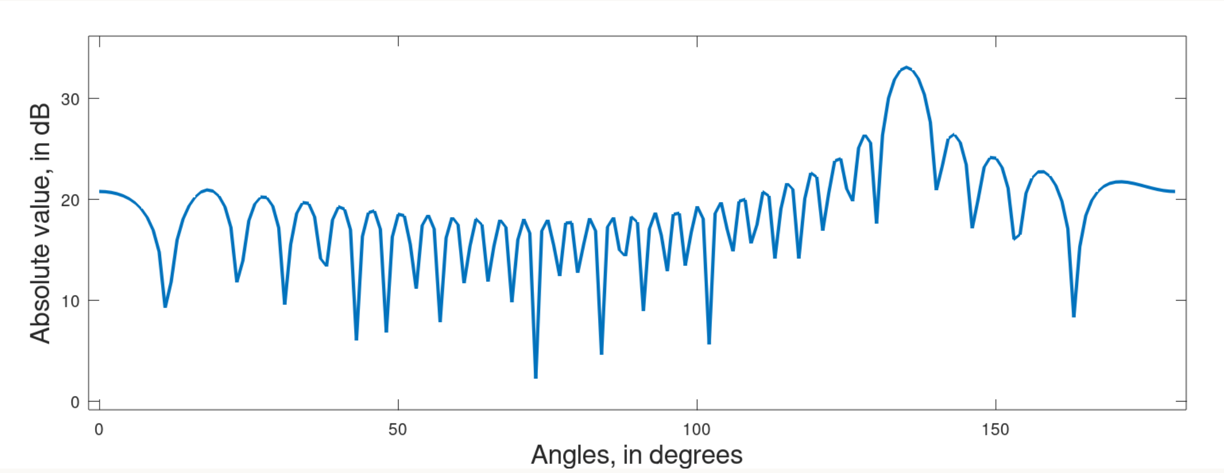

FIGURE 3.3: Beam-pattern for beamformer at angle 135

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
```

```cpp
    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 examine the properties of the beamformer under consideration. A beamformer can be regarded as a spatial filter: it attenuates signals arriving from undesired directions while providing gain to signals originating from the desired direction specified during the design process. This behavior is demonstrated by plotting the beamformer's gain as a function of angle.

4.2 Plots

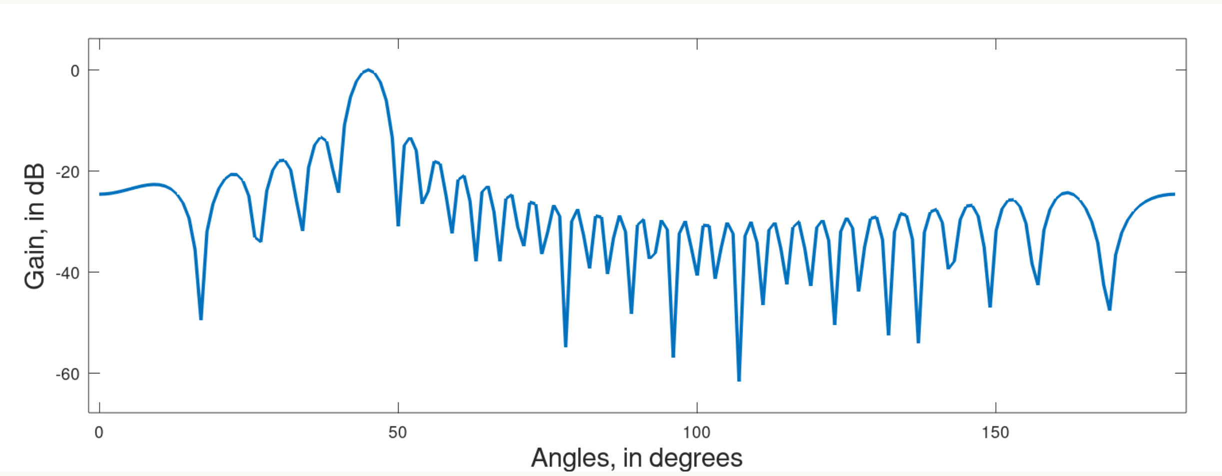

FIGURE 4.1: Beam-pattern for beamformer at angle 45

4.3 Observation

It can be observed that when the steering angle is modified, the beamformer's gain and attenuation shift correspondingly to the newly specified angle.

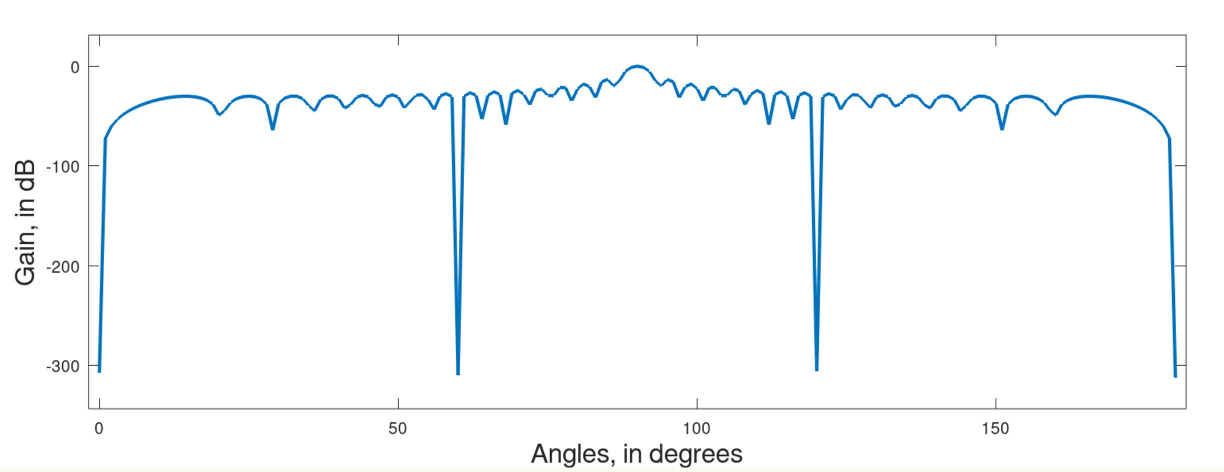

FIGURE 4.2: Beam-pattern for beamformer at angle 90

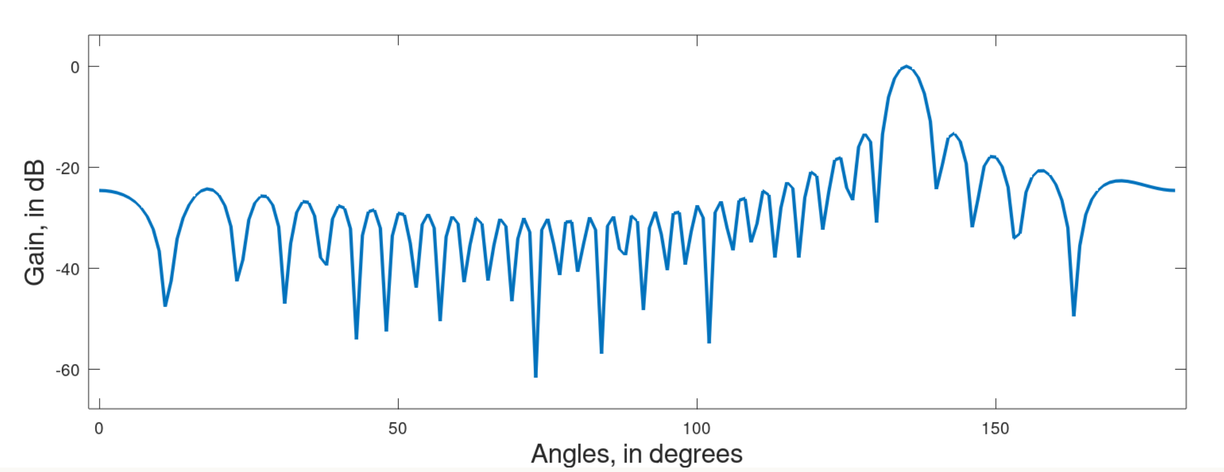

FIGURE 4.3: Beam-pattern for beamformer at angle 135

4.4 C++ Code

```
// ============================================================================
#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

The design of a linear array for a given frequency involves placing the array elements at an inter-element spacing equal to half the wavelength of the signal under consideration. As the spacing increases beyond this limit, the likelihood of end-fire anomalies also increases. An end-fire anomaly refers to the occurrence of additional maxima in the beam pattern, apart from the intended main lobe.

In our case, the array was designed for an operating frequency of 2 kHz. To investigate how variations in frequency influence the beam pattern, the procedure involves updating the weight vector to the corresponding frequency-dependent values and plotting the absolute beam response as a function of angle.

5.2 Plots

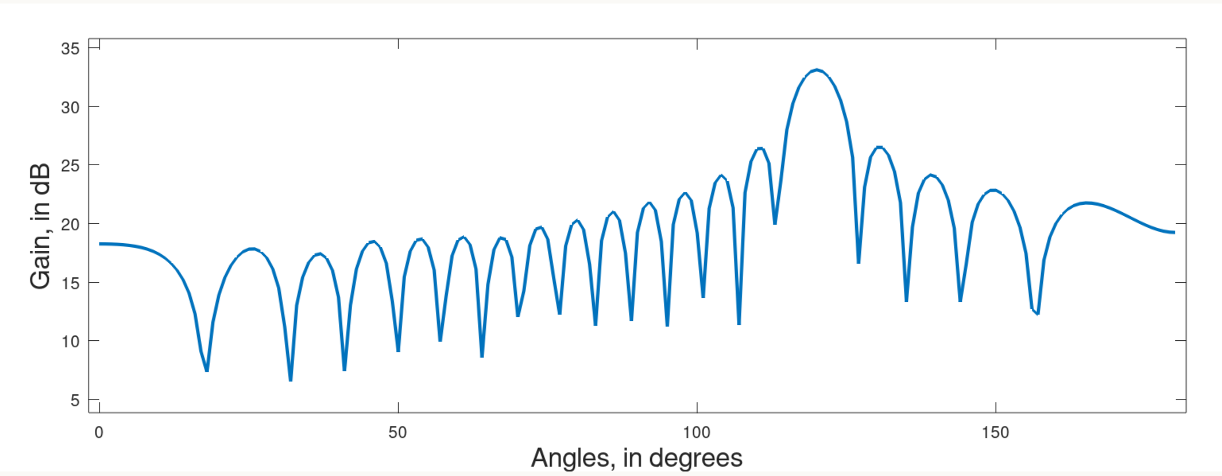

FIGURE 5.1: Beamformed for 1200Hz

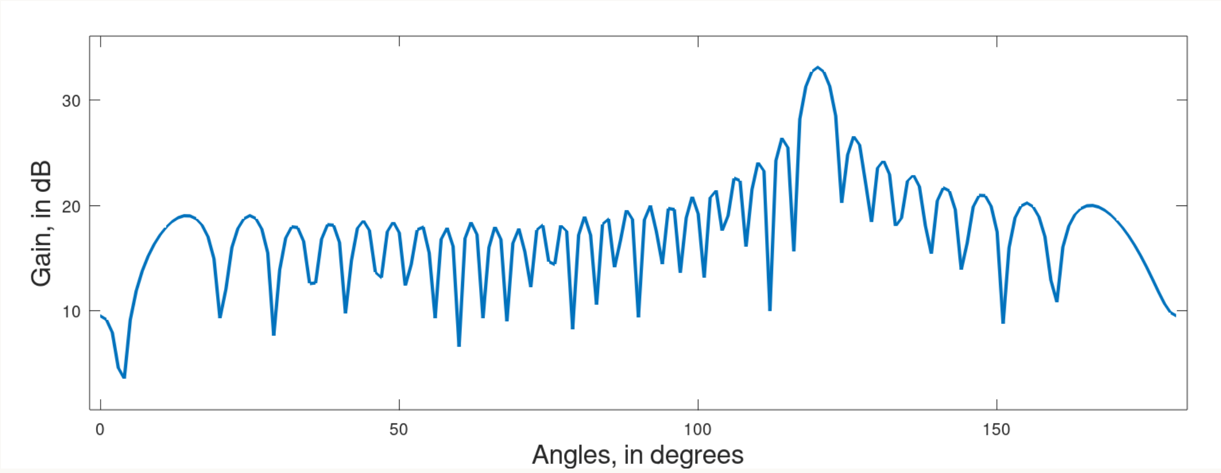

FIGURE 5.2: Beamformed for 2000Hz

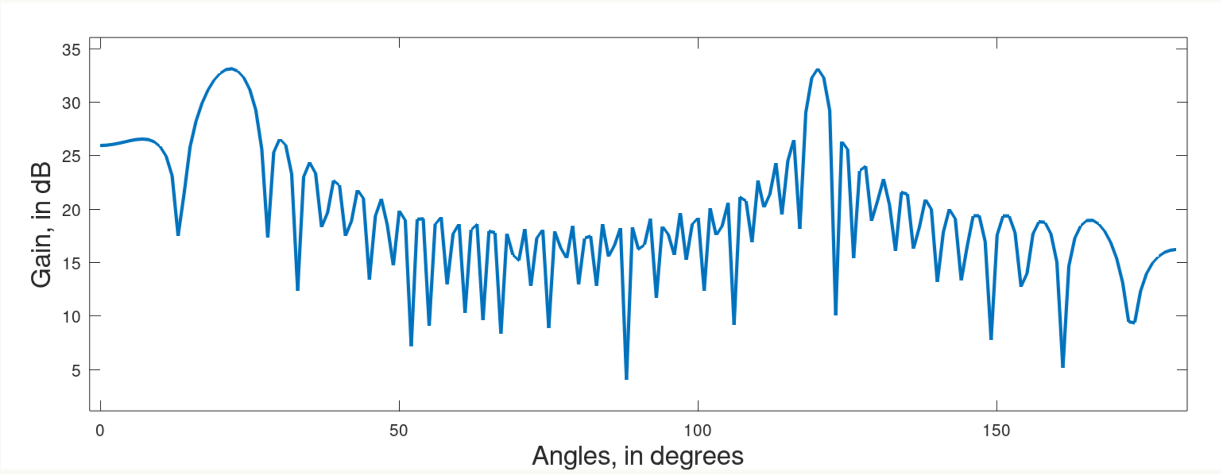

FIGURE 5.3: Beamformed for 2800Hz

5.3 Observation

It is observed that when the operating frequency falls below the designed frequency, the array continues to produce a valid beam pattern without the occurrence of end-fire anomalies. However, as the frequency increases beyond the design frequency, additional maxima appear in the beam pattern. This phenomenon introduces ambiguity in the directional response of the array.

5.4 C++ Code

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

    // starting timer
```

```cpp
    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)};
```

```cpp

    // 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

```matlab
%% 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

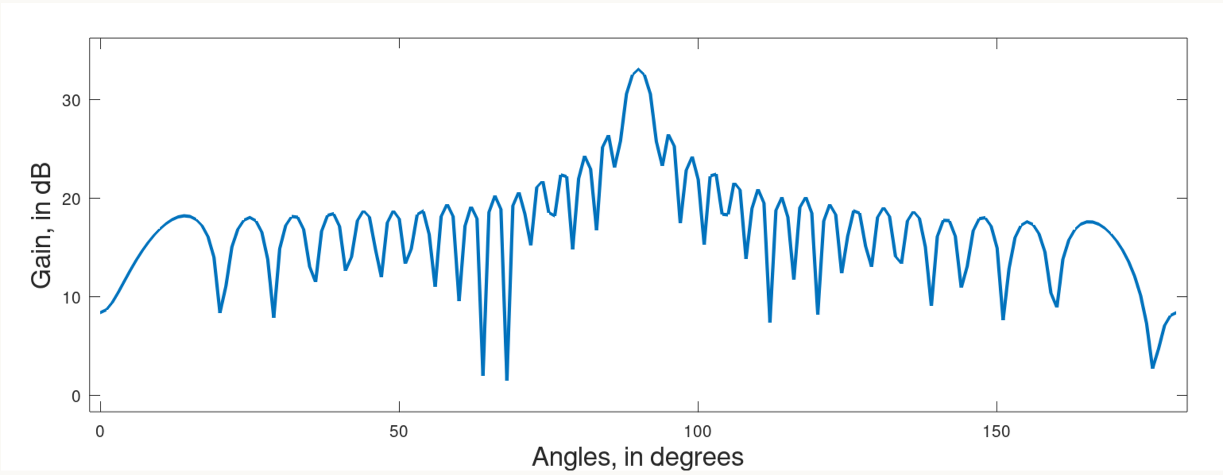

FIGURE 6.1: Beamforming for SNR = 10

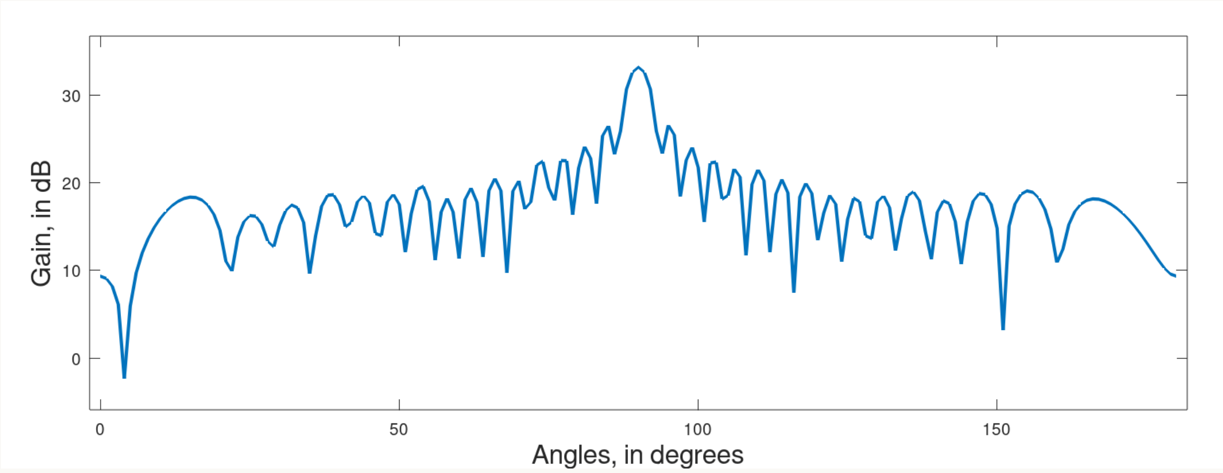

FIGURE 6.2: Beamforming for SNR = -1

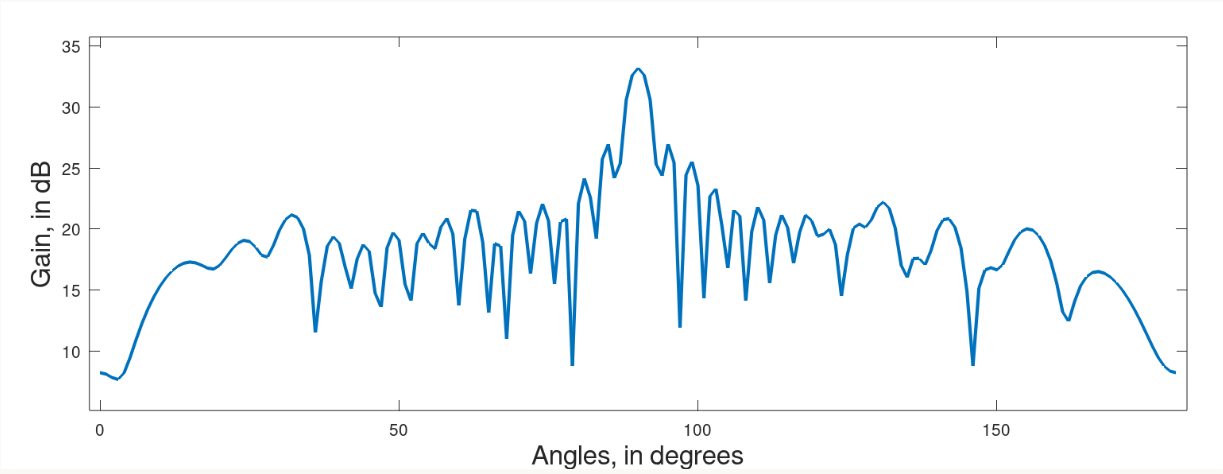

FIGURE 6.3: Beamforming for SNR = -10

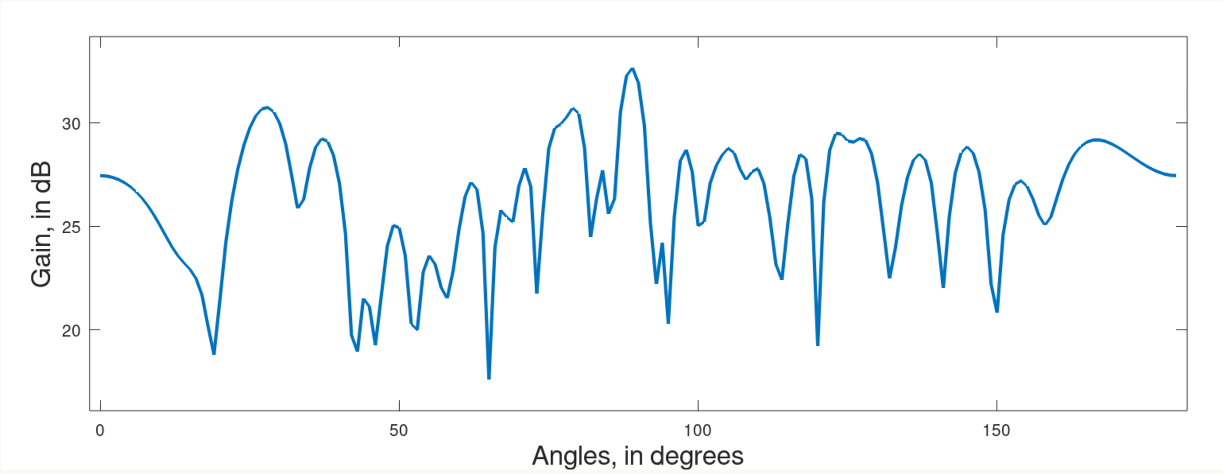

FIGURE 6.4: Beamforming for SNR = -30

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

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

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

    // 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

Simulate Broadband Beamforming

7.1 Aim

The objective is to perform beamforming on the incoming signal across different frequencies in order to estimate the direction of arrival (DOA) of the source. In this scenario, the signal originates from a single source at a single frequency; however, both the frequency and the angle of arrival are unknown. To address this, beamforming is applied over a range of candidate frequencies, while the steering angle is swept from 0 to 180 degrees.

For each frequency–angle pair, the absolute response is computed, and the results are aggregated to form an absolute-value-versus-angle plot. For simplicity, the analysis is restricted to nine discrete frequencies.

7.2 Plots

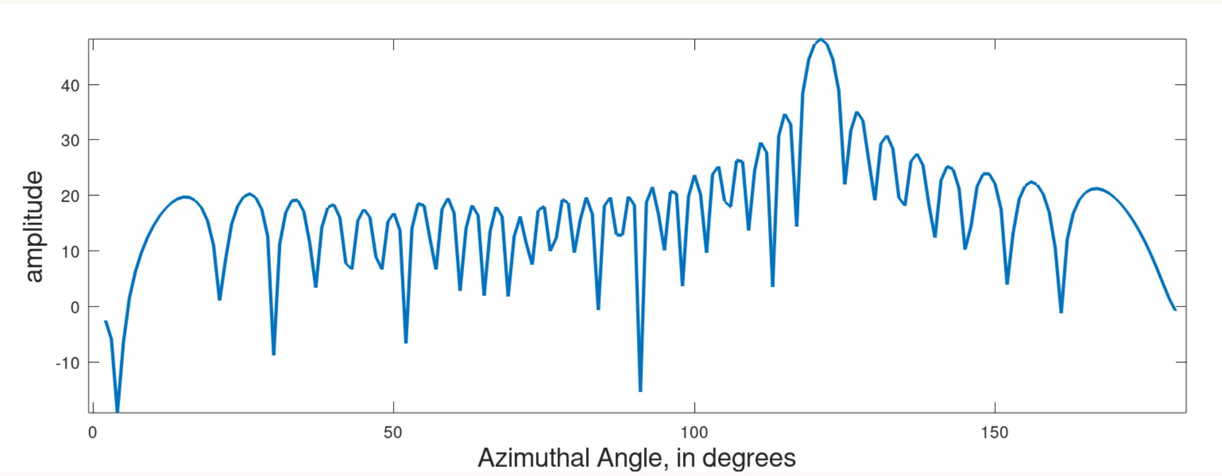

FIGURE 7.1

7.3 Observation

It can be observed that the resulting plot closely resembles the case of beamforming at a single frequency. For frequencies other than that of the source signal, the magnitude response remains negligible. A distinct peak emerges at the angle corresponding to the source location, while the responses at all other angles remain significantly suppressed.

7.4 C++ Code

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

    // starting timer
    auto logfile    {string("../csv-files/logfile-Objective8.csv")};
    Timer timer(logfile);

    // init-variables
    auto angle                {120};
    auto f                    {2000};
    auto Fs                   {12800};
    auto Ts                   {1.00/static_cast<double>(Fs)};
    auto c                    {1500};

    auto m                    {32};
    auto N                    {256};
    auto t                    {linspace(0.0, (N-1)*Ts, N)};

    auto lambda               {static_cast<double>(c)/2000};
    auto x                    {lambda/2.00};
    auto d                    {x * cosd(static_cast<double>(angle))/static_cast<double>(c)};
    auto matrix               {Zeros({m, N})};

    // far-field signal simulation
    auto& xaxis               {t};
    for(auto sensor_index = 0; sensor_index < m; ++sensor_index)
        matrix[sensor_index]    = \
            sin(2.00 * std::numbers::pi * static_cast<double>(f) * (t - sensor_index * d));

    // adding gaussian noise to sensor-outputs
    auto snr                  {2.00};
    auto snr_weight           {std::pow(10, -1 * snr * 0.05)};
    auto new_mat              {matrix + snr_weight * rand({m, N})};

    // Performing a basis-change
    auto& nfft                {N};
    auto fend                 {static_cast<double>((nfft-1)*Fs)/static_cast<double>(nfft)};
    auto waxis                {linspace(0.00, fend, nfft)};
    auto Fourier              {fft(new_mat, nfft)};
```

```
    // Beamforming
    auto delay_column       {vector<complex<double>>(m,
    static_cast<complex<double>>(0.00))};
    auto frequency_inter    {Fs/N};
    auto f_mat              {vector<complex<double>>(m, 0.00)};
    auto angle_matrix       {Zeros({N, 181})};
    auto frequency_matrix   {Zeros({9, 180})};
    auto index              {0};

    for(auto sweep_angle = 1; sweep_angle < 181; ++sweep_angle){
        for(auto f_sweep = 1000.00; f_sweep <= 3000.00; f_sweep += 1.00){

            // extracting frequency-indices
            index    = static_cast<double>(f * N)/static_cast<double>(Fs);
            auto temp    = slice(Fourier, {-1, -2, index, index});
            std::transform(temp.begin(), temp.end(),
                           f_mat.begin(),
                           [](auto argx){return argx[0];});

            // building delay-vector
            for(auto sensor_index = 0; sensor_index < m ; ++sensor_index)
                delay_column[sensor_index]  = \
                    std::exp(1i * sensor_index * 2.00 * \
                             std::numbers::pi * f * (x/c) * \
                             cosd(sweep_angle));

            // writing to frequency-matrix
            auto row_target     {static_cast<size_t>(f/250) - 3};
            frequency_matrix[row_target][sweep_angle]    = \
                std::abs(std::inner_product(f_mat.begin(),
                                            f_mat.end(),
                                            delay_column.begin(),
                                            complex<double>(0.00),
                                            std::plus<complex<double>>(),
                                            [](auto argx, auto argy){
                                                return argx * argy;
                                            }));
        }
    }

    // saving assets
    auto angle_axis     {linspace(1, 180, 180)};
    auto sum_matrix     {sum<0>(frequency_matrix)};
    fWriteVector(angle_axis, "../csv-files/angle_axis-Objective8.csv");
    fWriteVector(sum_matrix, "../csv-files/sum_matrix-Objective8.csv");

    // return
    return(0);

}
```

7.5 Octave Code

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

%% Loading the files
angle_axis      = csvread("../csv-files/angle_axis-Objective8.csv");
sum_matrix      = csvread("../csv-files/sum_matrix-Objective8.csv");
sum_matrix      = 20 * log10(sum_matrix);

%% Plotting
plotwidth   = 1515;
plotheight  = 500;

figure(1)
set(gcf, 'Position', [0 0 plotwidth plotheight]);   % [left bottom width height]
plot(angle_axis, sum_matrix, "LineWidth", 2);

xlim([min(angle_axis) - 1e-2 * range(angle_axis), max(angle_axis) + 1e-2 *
    range(angle_axis)]);
ylim([min(sum_matrix) - 1e-1 * range(sum_matrix), max(sum_matrix) + 1e-1 *
    range(sum_matrix)]);

xlabel("Azimuthal Angle, in degrees",  "fontsize", 16);
ylabel("amplitude",          "fontsize", 16);
saveas(gcf, "../Figures/sum-matrix-Objective8.png");
```

Chapter 8

Beamforming with Chebyshev Windowing

8.1 Aim

Traditional beamforming techniques, such as rectangular windowing, exhibit inherent limitations. Specifically, the first-, second-, and third-order sidelobes are suppressed by only 13.5, 18, and 21 dB, respectively, relative to the peak of the main lobe. Consequently, strong signals may be detected not only within the main lobe at the correct bearing but also through the sidelobes of adjacent beams. This effect introduces bearing ambiguities and spurious detections, thereby complicating subsequent processing tasks.

To mitigate these challenges, the design objective is to achieve the narrowest possible main lobe while maintaining sidelobe levels within an acceptable range. One approach is the use of Chebyshev windowing, where each array element's output is weighted by coefficients generated through the chebwin function in MATLAB.

8.2 Plots

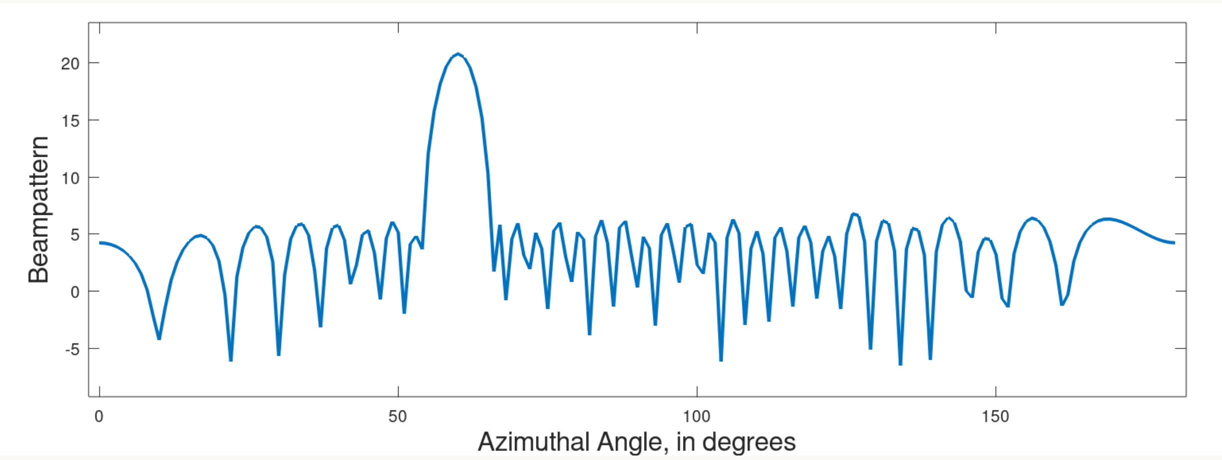

FIGURE 8.1

8.3 Observation

In this case, the sidelobes are uniformly suppressed to an equal level. However, this suppression is achieved at the expense of an increased main-lobe width, resulting in a broader central beam. Such a trade-off is particularly advantageous in applications where sidelobe reduction is critical, and a moderate widening of the main lobe can be tolerated.

8.4 C++ Code

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

    // starting timer
    auto logfile    {string("../csv-files/logfile-Objective9.csv")};
    Timer timer(logfile);

    // Initializing the variables
    auto f                          {2000.00};
    auto Fs                         {12800.00};
    auto Ts                         {1.00/Fs};
    auto N                          {128};
    auto m                          {32};
    auto angle                      {60.00};
    auto c                          {1500.00};
    auto lambda                     {c/f};
    auto x                          {lambda/2};
    auto d                          {x * cosd(angle)/c};
    auto snr                        {4};
    auto snr_weight                 {std::pow(10, -1 * snr * 0.05)};
    auto t                          {linspace(0.00, (N-1)*Ts, N)};
    auto matrix                     {Zeros({m, N})};


    // ULA signal simulation
    auto y              {sin(2 * std::numbers::pi * f * t)};
    for(auto sensor_index = 0; sensor_index < m; ++sensor_index){
        matrix[sensor_index] = sin(2.00 * std::numbers::pi * f * (t -
    static_cast<double>(sensor_index) * d));
    }
    auto new_mat        {matrix + snr_weight * rand({m, N})};


    // Moving signal to fourier-bases
    auto& nfft      {N};
    auto fend       {static_cast<double>((nfft-1)*Fs/nfft)};
    auto waxis      {linspace(0.00, fend, nfft)};
    auto Fourier    {fft(new_mat, nfft)};


```

```
    // Choosing the frequency row
    auto index      {static_cast<int>(static_cast<double>(f *
    nfft)/static_cast<double>(Fs))};
    auto f_mat      {slice(Fourier, {-1, -2, index, index})};


    // Building and applying chebyshev window
    auto cheb_window    {vector<double>{0.4439, 0.2433, 0.3035, 0.3684, 0.4367,
                                        0.5072, 0.5785, 0.6490, 0.7170, 0.7809,
                                        0.8392, 0.8904, 0.9330, 0.9660, 0.9886,
                                        1.0000, 1.0000, 0.9886, 0.9660, 0.9330,
                                        0.8904, 0.8392, 0.7809, 0.7170, 0.6490,
                                        0.5785, 0.5072, 0.4367, 0.3684, 0.3035,
                                        0.2433, 0.4439}};

    std::transform(f_mat.begin(), f_mat.end(),
                   cheb_window.begin(),
                   f_mat.begin(),
                   [](auto argx, auto argy){
                    return vector<complex<double>>{argx[0] * argy};
                   });


    // Beamforming
    auto angle_matrix   {vector<double>(181, 0)};
    auto delay_column   {vector<complex<double>>(32, 0)};

    for(auto test_angle = 0; test_angle < 181; ++test_angle){

        auto test_d         {x * cosd(test_angle)/c};

        for(auto sensor_index = 0; sensor_index < m; ++sensor_index)
            delay_column[sensor_index]  = \
                std::exp(1i * 2.00 * std::numbers::pi * f * sensor_index * test_d);

        angle_matrix[test_angle] = \
            std::abs(std::inner_product(f_mat.begin(), f_mat.end(),
                                        delay_column.begin(),
                                        complex<double>(0),
                                        std::plus<complex<double>>(),
                                        [](auto argx, auto argy){
                                            return argx[0] * argy;
                                        }));
    }


    // saving the tensors
    auto angle_axis     {linspace(0.0, 180.0, 181)};
    fWriteVector(angle_axis,    "../csv-files/angle-axis-Objective9.csv");
    fWriteVector(angle_matrix,  "../csv-files/angle-matrix-Objective9.csv");


    // return
    return(0);

}
```

8.5 Octave Code

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

%% Loading the files
angleaxis       = csvread("../csv-files/angle-axis-Objective9.csv");
angleMatrix     = 10 * log10(csvread("../csv-files/angle-matrix-Objective9.csv"));

%% Plotting
plotwidth   = 1515;
plotheight  = 500;

figure(1);
set(gcf, 'Position', [0 0 plotwidth plotheight]);   % [left bottom width height]
plot(angleaxis, angleMatrix, "LineWidth",2);
xlim([min(angleaxis)        - 1e-2 * range(angleaxis),
      max(angleaxis)        + 1e-2 * range(angleaxis)]);
ylim([min(angleMatrix)      - 1e-1 * range(angleMatrix),
      max(angleMatrix)      + 1e-1 * range(angleMatrix)]);
xlabel("Azimuthal Angle, in degrees",   "fontsize", 16);
ylabel("Beampattern",                   "fontsize", 16);
saveas(gcf, "../Figures/BeamPattern-Octave9.png");
```

Appendix A

C++ Function Definitions

The following sections present C++ functions and supporting code developed to implement the methods described in this report. As the implementation relies primarily on standard C++ and the Standard Template Library (STL), many tensor operations were manually constructed. Although mature libraries such as LibTorch and Boost are available, the decision was made to implement these components independently in order to gain proficiency in writing templated C++ code and to leverage the advantages of generic programming.

Template Metaprogramming (TMP) is a C++ paradigm in which partially specialized code is processed by the compiler to generate optimized, specialized implementations. Beyond reducing code redundancy and programmer effort, TMP enables certain computations to be shifted from runtime to compile time, thereby improving efficiency.

The principal motivation for adopting a templated approach in this work is rooted in the importance of generic programming in statically typed languages such as C++. Furthermore, in industrial contexts where systems often operate under real-time constraints, the ability to design efficient, generic, and templated code can play a critical role in reducing runtime latency.

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>
```

```cpp
#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 hashdefines.hpp

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

A.3 usings.hpp

```cpp
// 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.4 DataStructureDefinitions.hpp

```cpp
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.5 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.6 TimerClass.hpp

The Timer class is a lightweight utility designed to facilitate high-resolution performance measurement in C++ programs. It leverages the std::chrono library to capture precise timestamps at the start and end of code execution. The class maintains internal state variables to store time points (startpoint, endpoint) and their difference (duration), while also supporting optional metadata such as a log filename and a function name. This design allows users to associate timing data with specific program components, enabling both real-time performance monitoring and systematic data collection for later analysis.

The functionality of the class is structured around several key methods. The start() and stop() functions define the measurement interval, while the fetchtime() methods compute the elapsed time in nanoseconds and report it either directly or with an associated label. The measure() function provides a multi-resolution view of elapsed time, simultaneously reporting results in nanoseconds, microseconds, and seconds. This hierarchical reporting allows developers to evaluate both fine-grained and coarse-grained performance characteristics. Additionally, the class constructors are overloaded to support flexible initialization, allowing the user to specify optional log files or function identifiers at the time of instantiation.

The destructor (Timer) is designed to automatically compute and report the elapsed time upon object destruction, ensuring that timing information is not overlooked. Beyond console output, the destructor also supports persistent storage by writing timing data to a file if a filename is specified. This logging capability is particularly useful in larger systems where repeated profiling of functions is required, or where execution data must be aggregated for offline performance analysis. The inclusion of both immediate reporting and persistent logging highlights the class's utility as a practical tool for benchmarking and latency assessment in performance-critical applications.

```cpp
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;

    // 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";
                }
            }
        }
    }
};
```

A.7 utils.hpp

```
// ============================================================================
#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"
// =============================================================================
#include "svr_sum.hpp"
```

A.8 svr_WriteToCSV.hpp

The provided scripts implement generic C++ functions for writing vectors and matrices to external files, with support for both real and complex data types. The fWriteVector function is templated to handle vectors of arbitrary types. For real-valued types, the elements are written sequentially into a file with comma separation. For complex types (e.g., std::complex<float>, std::complex<double>, std::complex<long double>), the real and imaginary parts are explicitly written in the form a+bi, ensuring that complex-valued data is preserved in a human-readable format. These functions allow users to easily serialize numerical data for inspection, analysis, or further processing outside the C++ environment.

Similarly, the fWriteMatrix function provides templated functionality for writing two-dimensional matrices stored as nested std::vector objects. For general types, each row of the matrix is written as a comma-separated line, while a specialized template explicitly handles std::complex<double>, formatting entries in the a+bi representation. The functions incorporate basic error handling by checking file accessibility and reporting failures. Together, these implementations form a lightweight, extensible framework for exporting structured numerical data to files, thereby facilitating interoperability with other tools and supporting reproducibility in computational workflows.

```cpp
// ==============================================================================
template <typename T>
void fWriteVector(const vector<T>&              inputvector,
                  const string&                 filename){

    // opening a file
    std::ofstream fileobj(filename);
    if (!fileobj)   {return;}

    // writing the real parts in the first column and the imaginary parts int he
    second column
    if constexpr(std::is_same_v<T, std::complex<double>>    ||
                 std::is_same_v<T, std::complex<float>>     ||
                 std::is_same_v<T, std::complex<long double>>){
        for(int i = 0;  i<inputvector.size(); ++i){
            // adding entry
            fileobj << inputvector[i].real() << "+" << inputvector[i].imag() << "i";

            // adding delimiter
            if(i!=inputvector.size()-1) {fileobj << ",";}
            else                        {fileobj << "\n";}
        }
    }
    else{
        for(int i = 0; i<inputvector.size(); ++i){
            fileobj << inputvector[i];
            if(i!=inputvector.size()-1) {fileobj << ",";}
            else                        {fileobj << "\n";}
        }
    }

    // return
    return;
}
// Matrix writing ===============================================================
template <typename T>
auto fWriteMatrix(const std::vector<std::vector<T>> inputMatrix,
                  const string                      filename){

    // opening a file
    std::ofstream fileobj(filename);

```

```
    // writing
    if (fileobj){
        for(int i = 0; i<inputMatrix.size(); ++i){
            for(int j = 0; j<inputMatrix[0].size(); ++j){
                fileobj << inputMatrix[i][j];
                if (j!=inputMatrix[0].size()-1)     {fileobj << ",";}
                else                                {fileobj << "\n";}
            }
        }
    }
    else{
        cout << format("File-write to {} failed\n", filename);
    }

}

template <>
auto fWriteMatrix(const std::vector<std::vector<std::complex<double>>>  inputMatrix,
                  const string                                           filename){

    // opening a file
    std::ofstream fileobj(filename);

    // writing
    if (fileobj){
        for(int i = 0; i<inputMatrix.size(); ++i){
            for(int j = 0; j<inputMatrix[0].size(); ++j){
                fileobj << inputMatrix[i][j].real() << "+" <<
    inputMatrix[i][j].imag() << "i";
                if (j!=inputMatrix[0].size()-1)     {fileobj << ",";}
                else                                {fileobj << "\n";}
            }
        }
    }
    else{
        cout << format("File-write to {} failed\n", filename);
    }
}
```

A.9 svr_repmat.hpp

```
template<typename T>
constexpr auto repmat(const vector<vector<T>>&    input,
            const vector<int>           dimensions){

    // calculating resulting dimensions
    auto numrows    {static_cast<int>(input.size())     * dimensions[0]};
    auto numcols    {static_cast<int>(input[0].size())  * dimensions[1]};

    // creating new matrix
    vector<vector<T>>   finaloutput;
    vector<T>         temp;
    auto sourcerow  {-1};
```

```
    auto sourcecol  {-1};
    for(int i = 0; i<numrows; ++i){
        temp.clear();
        for(int j = 0; j<numcols; ++j){
            sourcerow = i % static_cast<int>(input.size());
            sourcecol = j % static_cast<int>(input[0].size());
            temp.push_back(input[sourcerow][sourcecol]);
        }
        finaloutput.push_back(temp);
    }

    // returning the final output
    return finaloutput;

};

template <typename T>
constexpr auto repmat(const vector<T>&            input,
            const vector<int>           dimensions){

    // calculating resulting dimensions
    auto numrows    {static_cast<int>(dimensions[0])};
    auto numcols    {static_cast<int>(input.size())  * dimensions[1]};

    // creating new matrix
    vector<vector<T>>   finaloutput;
    vector<T>           temp;

    // filling up the vector
    auto sourcerow  {-1};
    auto sourcecol  {-1};
    for(int i = 0; i<numrows; ++i){
        temp.clear();
        for(int j = 0; j<numcols; ++j){
            sourcerow = i % 1;
            sourcecol = j % static_cast<int>(input.size());
            temp.push_back(input[sourcecol]);
        }
        finaloutput.push_back(temp);
    }

    // returning the final output
    return finaloutput;

};
```

A.10 svr_linspace.hpp

```
// in-place
template <typename T>
auto linspace(auto&            input,
              auto             startvalue,
              auto             endvalue,
```

```
              auto            numpoints) -> void
{
    auto stepsize = static_cast<T>(endvalue -
    startvalue)/static_cast<T>(numpoints-1);
    for(int i = 0; i<input.size(); ++i) {input[i] = startvalue + i*stepsize;}
};
// in-place
template <typename T>
auto linspace(vector<complex<T>>&   input,
              auto                  startvalue,
              auto                  endvalue,
              auto                  numpoints) -> void
{
    auto stepsize = static_cast<T>(endvalue -
    startvalue)/static_cast<T>(numpoints-1);
    for(int i = 0; i<input.size(); ++i) {
        input[i] = startvalue + static_cast<T>(i)*stepsize;
    }
};

// return-type
template <typename T>
auto linspace(T             startvalue,
              T             endvalue,
              size_t        numpoints)
{
    vector<T> input(numpoints);
    auto stepsize = static_cast<T>(endvalue -
    startvalue)/static_cast<T>(numpoints-1);

    for(int i = 0; i<input.size(); ++i) {input[i] = startvalue +
    static_cast<T>(i)*stepsize;}

    return input;
};

// return-type
template <typename T, typename U>
auto linspace(T             startvalue,
              U             endvalue,
              size_t        numpoints)
{
    vector<double> input(numpoints);
    auto stepsize = static_cast<double>(endvalue -
    startvalue)/static_cast<double>(numpoints-1);

    for(int i = 0; i<input.size(); ++i) {input[i] = startvalue + i*stepsize;}

    return input;
};
```

A.11 svr_fft.hpp

```
// ==============================================================================
template<typename T>
auto fft(vector<T> inputarray){

    // building just the time thing
    vector<complex<T>> basiswithoutfrequency(inputarray.size(), 0);
    linspace(basiswithoutfrequency,
             0,
             basiswithoutfrequency.size()-1,
             basiswithoutfrequency.size());

    auto lambda = \
        [&basiswithoutfrequency](const complex<T> arg){
        return std::exp(-1.00 * \
                        std::complex<T>{0, 1} * \
                        2.00 * std::numbers::pi * \
                        static_cast<complex<T>>(arg) \
                        / static_cast<complex<T>>(basiswithoutfrequency.size()));
    };
    std::transform(basiswithoutfrequency.begin(),
                   basiswithoutfrequency.end(),
                   basiswithoutfrequency.begin(),
                   lambda);

    // building basis vectors
    vector<vector<complex<T>>> basisvectors;
    for(auto i = 0; i < inputarray.size(); ++i){

        // making a copy of the basis-without-frequency
        auto temp   = basiswithoutfrequency;

        // exponentiating with associated frequency
        std::transform(temp.begin(),
                       temp.end(),
                       temp.begin(),
                       [i](const auto arg1){
                        return std::pow(arg1, i);
                    });

        // pushing to end of basis vectors
        basisvectors.push_back(std::move(temp));
    }

    // building coefficient arrays
    vector<complex<double>> finaloutput(inputarray.size(), 0);
    finaloutput.reserve(finaloutput.size());

    // producing inner-products
    for(int i = 0; i<inputarray.size(); ++i){
        finaloutput[i] = \
            std::inner_product(basisvectors[i].begin(),
                               basisvectors[i].end(),
                               inputarray.begin(),
                               complex<double>(0),
                               std::plus<std::complex<double>>(),
                               [&inputarray](complex<double> a, T b){
                                   return a * static_cast<complex<double>>(b) /
    static_cast<double>(std::sqrt(inputarray.size()));
```

```
                                });
    }

    // returning finaloutput
    return finaloutput;
}
// ==============================================================================
template<typename T>
auto fft(vector<T> inputarray, size_t nfft){

    // throwing an error
    if (nfft < inputarray.size())   {std::cerr << "size-mistmatch\n";}

    // building time-only basis
    vector<complex<T>>
    basiswithoutfrequency = linspace(static_cast<std::complex<T>>(0),
                                     static_cast<std::complex<T>>(nfft-1),
                                     nfft);
    auto lambda = [&basiswithoutfrequency](const complex<T> arg){
        return std::exp(-1.00 * complex<double>{0, 1} * 2.00 * \
            std::numbers::pi * static_cast<complex<T>>(arg) / \
            static_cast<complex<T>>(basiswithoutfrequency.size()));
    };
    std::transform(basiswithoutfrequency.begin(),
                   basiswithoutfrequency.end(),
                   basiswithoutfrequency.begin(),
                   lambda);


    // building basis vectors
    vector<vector<complex<double>>> basisvectors;
    for(auto i = 0; i < inputarray.size(); ++i){

        // making a copy of the basis-without-frequency
        vector<complex<double>> temp = basiswithoutfrequency;

        // exponentiating with associated frequency
        std::transform(temp.begin(),
                       temp.end(),
                       temp.begin(),
                       [i](const auto arg1){return std::pow(arg1, i);});

        // pushing to end of basis vectors
        basisvectors.push_back(std::move(temp));
    }


    // building the projection
    vector<complex<double>> finaloutput(inputarray.size(), 0);
    finaloutput.reserve(finaloutput.size());
    #pragma omp parallel for
    for(int i = 0; i<inputarray.size(); ++i){
        // writing coefficients
        finaloutput[i] = \
            std::inner_product(basisvectors[i].begin(),
                               basisvectors[i].end(),
                               inputarray.begin(),
                               complex<double>(0),
```

```
                                std::plus<std::complex<double>>(),
                                [&nfft](const complex<double>   a,
                                   const T                 b){
                                     return a * static_cast<complex<double>>(b) /
    static_cast<double>(std::sqrt(nfft));
                                });
    }

    // returning finaloutput
    return finaloutput;
}

// ================================================================================
template <>
auto fft(vector<complex<double>> inputarray){

    // aliasing
    using T = double;

    // building time-only basis-vector
    vector<complex<T>>
    basiswithoutfrequency = linspace(std::complex<T>(0),
                                      std::complex<T>(inputarray.size()-1),
                                      inputarray.size());
    auto lambda = [&basiswithoutfrequency](const complex<T> arg){
        return std::exp(-1.00 * complex<double>{0, 1} * 2.00 * \
                        std::numbers::pi * static_cast<complex<T>>(arg) \
                        / static_cast<complex<T>>(basiswithoutfrequency.size()));
    };
    std::transform(basiswithoutfrequency.begin(),
                   basiswithoutfrequency.end(),
                   basiswithoutfrequency.begin(),
                   lambda);


    // building basis vectors
    vector<vector<complex<T>>> basisvectors;
    for(auto i = 0; i < inputarray.size(); ++i){

        // making a copy of the basis-without-frequency
        vector<complex<T>> temp = basiswithoutfrequency;

        // adding frequency component
        std::transform(temp.begin(),
                       temp.end(),
                       temp.begin(),
                       [i](const auto arg1){return std::pow(arg1, i);});

        // pushing to end of basis vectors
        basisvectors.push_back(std::move(temp));
    }


    // building the coefficients
    vector< complex<T> > finaloutput(inputarray.size(), 0);
    finaloutput.reserve(finaloutput.size());
    for(int i = 0; i<inputarray.size(); ++i)
        finaloutput[i] = std::inner_product(basisvectors[i].begin(),
```

```
                                                                basisvectors[i].end(),
                                                                inputarray.begin(),
                                                                std::complex<double>(0));

    // scaling down the coefficients
    std::transform(finaloutput.begin(),
                   finaloutput.end(),
                   finaloutput.begin(),
                   [&inputarray](auto argx){
                        return argx / static_cast<T>(std::sqrt(inputarray.size()));
                   });


    // returning finaloutput
    return finaloutput;
}

// ================================================================================
template<typename T>
auto fft(std::vector<std::vector<T>> inputMatrix,
         int                         nfft){

    // initializing
    std::vector<std::vector<std::complex<T>>>
    finaloutput(inputMatrix.size(),
                std::vector<std::complex<T>>(inputMatrix[0].size(), 0));

    // checking if we need to pad the rows
    if (inputMatrix[0].size() > nfft)   {std::cerr << "nfft < row-size\n";}
    else if (inputMatrix[0].size() < nfft)    {

        // creating a placeholder
        std::vector<std::vector<std::complex<T>>>
        temp(inputMatrix.size(),
             std::vector<std::complex<T>>(nfft, 0));

        // moving to the finaloutput
        finaloutput.clear();
        finaloutput = std::move(temp);

    }

    // filling final-output with the input-values
    for(int i = 0; i<inputMatrix.size(); ++i)
        std::copy(inputMatrix[i].begin(),
                  inputMatrix[i].end(),
                  finaloutput[i].begin());

    // performing fft
    #pragma omp parallel for
    for(auto& row: finaloutput)     {row = fft(row);}

    // returning the matrix
    return finaloutput;

}
// ================================================================================
template<>
```

```
auto fft(std::vector<std::vector<std::complex<double>>> inputMatrix,
         int                                                nfft){

    // changing types
    using T = double;

    // initializing
    std::vector<std::vector<std::complex<T>>>
    finaloutput(inputMatrix.size(),
                std::vector<std::complex<T>>(inputMatrix[0].size(), complex<T>(0)));

    // checking if we need to pad the rows
    if (inputMatrix[0].size() > nfft)   {std::cerr << "nfft < row-size\n";}
    else if (inputMatrix[0].size() < nfft)    {

        // creating a placeholder
        std::vector<std::vector<std::complex<T>>>
        temp(inputMatrix.size(),
             std::vector<std::complex<T>>(nfft, std::complex<T>(0)));

        // moving to the finaloutput
        finaloutput.clear();
        finaloutput = std::move(temp);
    }

    // filling final-output with the input-values
    for(int i = 0; i<inputMatrix.size(); ++i){
        std::copy(inputMatrix[i].begin(),
                  inputMatrix[i].end(),
                  finaloutput[i].begin());
    }

    // performing fft
    for(int i = 0; i<finaloutput.size(); ++i)
        finaloutput[i]  = std::move(fft(finaloutput[i]));

    // returning the matrix
    return finaloutput;

}
// ifft (vector) ===============================================================
template <typename T>
auto ifft(const std::vector<T> inputvector){

    // setting up data type
    using T2 = std::conditional_t<std::is_same_v<T, std::complex<float>>,
                                  std::complex<float>,
                                  std::complex<double>>;
    using T3 = typename T2::value_type;

    // building basis
    vector<T2>
    basiswithoutfrequency   {linspace(static_cast<T2>(0),
                                      static_cast<T2>(inputvector.size()-1),
                                      inputvector.size())};

    // lambda for building basis without frequency
    auto lambda = \
```

```
        [&basiswithoutfrequency](const T2 arg){
            return std::exp(1.00 * T2{0, 1} * 2.00 * \
                            std::numbers::pi * arg / \
                            static_cast<T2>(basiswithoutfrequency.size()));
    };

    // building the basis without frequency
    std::transform(basiswithoutfrequency.begin(),
                   basiswithoutfrequency.end(),
                   basiswithoutfrequency.begin(),
                   lambda);

    // building basis vectors
    std::vector<std::vector<T2>> bases;
    for(int i = 0; i < inputvector.size(); ++i){

        // creating bases with frequency components
        auto basiswithfrequency = basiswithoutfrequency;
        std::transform(basiswithfrequency.begin(),
                       basiswithfrequency.end(),
                       basiswithfrequency.begin(),
                       [i](T2 argx){
                            return static_cast<T2>(std::pow(argx,
    static_cast<T3>(i)));
                       });

        // pushing to the basis vectors
        bases.push_back(std::move(basiswithfrequency));
    }

    // computing projections
    std::vector<T2> projection_coefficients(inputvector.size());
    for(auto i = 0; i < bases.size(); ++i){

        // calculating inner-product
        auto temp   {std::inner_product(bases[i].begin(), bases[i].end(),
                                        inputvector.begin(),
                                        static_cast<T2>(0),
                                        std::plus<T2>(),
                                        [&inputvector](auto arg_bases, auto
    arg_inputvector){
                                            return static_cast<T2>(arg_bases) *
        \
                                                   static_cast<T2>(arg_inputvector)
    /   \

    static_cast<T3>(std::sqrt(inputvector.size()));
                                        })};

        // writing to the final output
        projection_coefficients[i] = std::move(temp);
    }

    // returning the coefficients
    return projection_coefficients;

}
```

A.12 svr_rand.hpp

```
// ===============================================================================
template <typename T>
auto rand(const T min, const T max) {
    static std::random_device rd;   // Seed
    static std::mt19937 gen(rd());  // Mersenne Twister generator
    std::uniform_real_distribution<> dist(min, max);
    return dist(gen);
}
// ===============================================================================
template <typename T>
auto rand(const T       min,
          const T       max,
          const size_t  numelements) {
    static std::random_device rd;   // Seed
    static std::mt19937 gen(rd());  // Mersenne Twister generator
    std::uniform_real_distribution<> dist(min, max);

    // building the fianloutput
    vector<T> finaloutput(numelements);
    for(int i = 0; i<finaloutput.size(); ++i)   {finaloutput[i] = static_cast<T>(dist(gen));}

    return finaloutput;
}
// ===============================================================================
template <typename T>
auto rand(const T               argmin,
          const T               argmax,
          const vector<int>     dimensions){

    // throwing an error if dimension is greater than two
    if (dimensions.size() > 2)  {std::cerr << "dimensions are too high\n";}

    // creating random engine
    static std::random_device rd;   // Seed
    static std::mt19937 gen(rd());  // Mersenne Twister generator
    std::uniform_real_distribution<> dist(argmin, argmax);

    // building the finaloutput
    vector<vector<T>> finaloutput;
    for(int i = 0; i<dimensions[0]; ++i){
        vector<T> temp;
        for(int j = 0; j<dimensions[1]; ++j)    {temp.push_back(dist(gen));}
        // cout << format("\t\t temp = {}\n", temp);

        finaloutput.push_back(temp);
    }

    // returning the finaloutput
    return finaloutput;

}
// ===============================================================================
auto rand(const vector<int>     dimensions){

```

```
    using ReturnType = double;

    // throwing an error if dimension is greater than two
    if (dimensions.size() > 2)  {std::cerr << "dimensions are too high\n";}

    // creating random engine
    static std::random_device rd;   // Seed
    static std::mt19937 gen(rd());  // Mersenne Twister generator
    std::uniform_real_distribution<> dist(0.00, 1.00);

    // building the finaloutput
    vector<vector<ReturnType>> finaloutput;
    for(int i = 0; i<dimensions[0]; ++i){
        vector<ReturnType> temp;
        for(int j = 0; j<dimensions[1]; ++j)    {temp.push_back(dist(gen));}
        finaloutput.push_back(std::move(temp));
    }

    // returning the finaloutput
    return finaloutput;

}
// ============================================================================
template <typename T>
auto rand_complex_double(const T                argmin,
                         const T                argmax,
                         const vector<int>&  dimensions){

    // throwing an error if dimension is greater than two
    if (dimensions.size() > 2)  {std::cerr << "dimensions are too high\n";}

    // creating random engine
    static std::random_device rd;   // Seed
    static std::mt19937 gen(rd());  // Mersenne Twister generator
    std::uniform_real_distribution<> dist(argmin, argmax);

    // building the finaloutput
    vector<vector<complex<double>>> finaloutput;
    for(int i = 0; i<dimensions[0]; ++i){
        vector<complex<double>> temp;
        for(int j = 0; j<dimensions[1]; ++j)
    {temp.push_back(static_cast<double>(dist(gen)));}
        finaloutput.push_back(std::move(temp));
    }

    // returning the finaloutput
    return finaloutput;

}
```

A.13 svr_operator_star.hpp

```
// scalar * vector ==============================================================
```

```
template <typename T>
auto operator*(T                       scalar,
               const vector<T>&     inputvector){
    vector<T> temp(inputvector.size());
    std::transform(inputvector.begin(),
                   inputvector.end(),
                   temp.begin(),
                   [&scalar](T x){return scalar * x;});
    return temp;
}

template <typename T1, typename T2>
auto operator*(T1                          scalar,
               const vector<T2>&        inputvector){
    using T3 = decltype(std::declval<T1>() * std::declval<T2>());
    vector<T3> temp(inputvector.size());
    std::transform(inputvector.begin(),
                   inputvector.end(),
                   temp.begin(),
                   [&scalar](auto x){return static_cast<T3>(scalar) *
    static_cast<T3>(x);});
    return temp;
}

// vector * scalar ==============================================================
template <typename T>
auto operator*(const vector<T>& inputvector,
               T                scalar){
    vector<T> temp(inputvector.size());
    std::transform(inputvector.begin(), inputvector.end(), temp.begin(),
    [&scalar](T x){return scalar * x;});
    return temp;
}
// scalar * matrix ==============================================================
template <typename T>
auto operator*(T                                    scalar,
               const std::vector<std::vector<T>>&   inputMatrix){
    std::vector<std::vector<T>> temp    {inputMatrix};
    for(int i = 0; i<inputMatrix.size(); ++i){
        std::transform(inputMatrix[i].begin(),
                       inputMatrix[i].end(),
                       temp[i].begin(),
                       [&scalar](T x){return scalar * x;});
    }
    return temp;
}
// scalar * matrix ==============================================================
template <typename T1, typename T2>
auto operator*(T1 scalar,
               const std::vector<std::vector<T2>>& inputMatrix){
    std::vector<std::vector<T2>> temp    {inputMatrix};
    for(int i = 0; i<inputMatrix.size(); ++i){
        std::transform(inputMatrix[i].begin(),
                       inputMatrix[i].end(),
                       temp[i].begin(),
                       [&scalar](T2 x){return static_cast<T2>(scalar) * x;});
    }
    return temp;
```

```cpp
}

// matrix * matrix ==========================================================
template <typename T>
auto operator*(const std::vector<std::vector<T>>& matA,
               const std::vector<std::vector<T>>& matB)
{

    // throwing error
    if (matA.size() != matB.size() || matA[0].size() != matB[0].size())
    {std::cerr << "size issues\n";}

    // creating placeholder
    auto temp   {matA};

    // performing multiplication
    for(int i = 0; i<matA.size(); ++i){
        for(int j = 0; j<matA[0].size(); ++j){
            temp[i][j] *= matB[i][j];
        }
    }

    // returning
    return temp;
}
// matrix-multiplication ====================================================
template <typename T1, typename T2>
auto matmul(const std::vector<std::vector<T1>>& matA,
            const std::vector<std::vector<T2>>& matB)
{

    // throwing error
    if (matA[0].size() != matB.size())      {std::cerr << "dimension-mismatch \n";}

    // getting result-type
    using ResultType    = decltype(std::declval<T1>() * std::declval<T2>() + \
                                   std::declval<T1>() * std::declval<T2>() );

    // creating aliasses
    auto finalnumrows  {matA.size()};
    auto finalnumcols  {matB[0].size()};

    // creating placeholder
    auto rowcolproduct  = [&](auto rowA, auto colB){
        ResultType temp   {0};
        for(int i = 0; i < matA.size(); ++i)    {temp +=
    static_cast<ResultType>(matA[rowA][i]) +
    static_cast<ResultType>(matB[i][colB]);}
        return temp;
    };

    // producing row-column combinations
    std::vector<std::vector<ResultType>> finaloutput(finalnumrows,
    std::vector<ResultType>(finalnumcols));
    for(int row = 0; row < finalnumrows; ++row){for(int col = 0; col <
    finalnumcols; ++col){finaloutput[row][col]   = rowcolproduct(row, col);}}

    // returning
```

```
    return finaloutput;
}
// ============================================================================
```

A.14 svr_operators.hpp

```
template <typename T>
std::vector<T> operator+(const std::vector<T>& a, const std::vector<T>& b) {
    // Identify which is bigger
    const auto& big = (a.size() > b.size()) ? a : b;
    const auto& small = (a.size() > b.size()) ? b : a;

    std::vector<T> result = big;  // copy the bigger one

    // Add elements from the smaller one
    for (size_t i = 0; i < small.size(); ++i) {
        result[i] += small[i];
    }

    return result;
}
template <typename T>
std::vector<T>& operator+=(std::vector<T>& a, const std::vector<T>& b) {

    const auto& small = (a.size() < b.size()) ? a : b;
    const auto& big   = (a.size() < b.size()) ? b : a;

    // If b is bigger, resize 'a' to match
    if (a.size() < b.size())                        {a.resize(b.size());}

    // Add elements
    for (size_t i = 0; i < small.size(); ++i)       {a[i] += b[i];}

    // returning elements
    return a;
}
template <typename T>
std::vector<std::vector<T>> operator+(const std::vector<std::vector<T>>& a,
                                      const std::vector<std::vector<T>>& b)
{
    // throwing an error if dimension error occurrs
    if ((a.size() != b.size()) || (a[0].size() != b[0].size()))           {
        cout << format("a.dimensions = [{},{}], b.shape = [{},{}]\n",
            a.size(), a[0].size(), b.size(), b[0].size());
        std::cerr << "dimensions don't match\n";
    }


    // performing the addition
    auto temp   {a};
    for(int i = 0; i<b.size(); ++i){
        for(int j = 0; j<b[0].size(); ++j){
            temp[i][j] += b[i][j];
```

```
        }
    }

    // retuerning
    return temp;
}
// ============================================================================
// Aim: substracting scalar from a vector
template <typename T>
std::vector<T> operator-(const std::vector<T>& a, const T scalar){
    std::vector<T> temp(a.size());
    std::transform(a.begin(),
                   a.end(),
                   temp.begin(),
                   [scalar](T x){return (x - scalar);});
    return temp;
}
// scalar operators ===========================================================
auto operator*(const std::complex<double>   complexscalar,
               const double                 doublescalar){
    return complexscalar * static_cast<std::complex<double>>(doublescalar);
}
auto operator*(const double                 doublescalar,
               const std::complex<double>   complexscalar){
    return complexscalar * static_cast<std::complex<double>>(doublescalar);
}
auto operator*(const std::complex<double>   complexscalar,
               const int                    scalar){
    return complexscalar * static_cast<std::complex<double>>(scalar);
}
auto operator*(const int                    scalar,
               const std::complex<double>   complexscalar){
    return complexscalar * static_cast<std::complex<double>>(scalar);
}
```

A.15 svr_tensor_inits.hpp

```
std::vector<std::vector<double>> Zeros(vector<int> dimensions){

    // throwing an error if the dimension is more than 2
    if (dimensions.size() > 2) {std::cerr << "Dimensions are a little too much";}

    // building the vector
    std::vector<std::vector<double>> temp;
    for(int i = 0; i<dimensions[0]; ++i){
        temp.emplace_back(vector<double>(dimensions[1], 0));
    };

    // returning the output
    return temp;
}

auto Zeros_complex_double(vector<int> dimensions){
```

```

    // throwing an error if the dimension is more than 2
    if (dimensions.size() > 2) {std::cerr << "Dimensions are a little too much";}

    // building the vector
    std::vector<std::vector<std::complex<double>>> temp;
    for(int i = 0; i<dimensions[0]; ++i){
        temp.emplace_back(std::vector<std::complex<double>>(dimensions[1],
    std::complex<double>{0,0}));
    };

    // returning the output
    return temp;
}

// ============================================================================
std::vector<std::vector<double>> Ones(vector<int> dimensions){

    // throwing an error if the dimension is more than 2
    if (dimensions.size() > 2) {std::cerr << "Dimensions are a little too much";}

    // building the vector
    std::vector<std::vector<double>> temp;
    for(int i = 0; i<dimensions[0]; ++i){
        temp.emplace_back(vector<double>(dimensions[1], 1));
    };

    // returning the output
    return temp;
}
```

A.16 svr_sin.hpp

```
template <typename T>
auto sin(vector<T> input){
    auto temp   {input};
    std::transform(input.begin(),
                   input.end(),
                   temp.begin(),
                   [](const T x){return std::sin(x);});
    return temp;
}

template <typename T>
auto sin_inplace(vector<T>& input) -> void
{
    std::transform(input.begin(),
                   input.end(),
                   input.begin(),
                   [](const T x){return std::sin(x);});
}

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

```cpp
template <typename T>
auto cosd(T input){
    return std::cos(input * std::numbers::pi / 180);
}
```

A.17 svr_slice.hpp

```cpp
template<typename T>
auto slice(const std::vector<std::vector<T>>&   inputMatrix,
           vector<int>                          arglist)
{

    // updating rows and columns
    if (arglist[0] == -1)       {arglist[0] = 0;}
    else if(arglist[0] == -2)   {arglist[0] = inputMatrix.size()-1;}
    if (arglist[1] == -1)       {arglist[1] = 0;}
    else if(arglist[1] == -2)   {arglist[1] = inputMatrix.size()-1;}
    if (arglist[2] == -1)       {arglist[2] = 0;}
    else if(arglist[2] == -2)   {arglist[2] = inputMatrix[0].size()-1;}
    if (arglist[3] == -1)       {arglist[3] = 0;}
    else if(arglist[3] == -2)   {arglist[3] = inputMatrix[0].size()-1;}

    // storing dimension values
    int rowsize    {arglist[1] - arglist[0] + 1};
    int colsize    {arglist[3] - arglist[2] + 1};

    // building the final-output matrix
    std::vector<std::vector<T>> finaloutput;
    for(int row = arglist[0]; row <= arglist[1]; ++row){

        // creating empty sub-row
        vector<T> temp(colsize, 0);

        // copying corrresponding region to subrow
        std::copy(inputMatrix[row].begin() + arglist[2],
                  inputMatrix[row].begin() + arglist[3]+1,
                  temp.begin());

        // pushing to final output
        finaloutput.push_back(std::move(temp));
    }

    // returning the final output
    return finaloutput;
}
```

A.18 svr_matrix_operations.hpp

```cpp
// ============================================================================
```

```
template <typename T>
auto dotproduct(const vector<T> argx,
                const vector<T> argy)
{
    // dimension checks
    if (argx.size() != argy.size()) {std::cerr << "size disparity\n";}

    // accumulating
    T   temp    = 0;
    for(int i = 0; i<argx.size(); ++i){

        if constexpr(std::is_same_v<T, std::complex<double>> ||
                     std::is_same_v<T, std::complex<float>> ||
                     std::is_same_v<T, std::complex<long double>>){
            temp += std::conj(argx[i]) * argy[i];
        }
        else{
            temp += argx[i] * argy[i];
        }
    }


    return temp;
}
```

A.19 svr_shape.hpp

```
// returning shape as vector ===================================================
template <typename T>
auto shape(const T& inputTensor){

    using U = std::decay_t<T>;

    // type-traits checking
    if constexpr (std::is_same_v<U, std::vector<double>>            ||
                  std::is_same_v<U, std::vector<float>>             ||
                  std::is_same_v<U, std::vector<int>>               ||
                  std::is_same_v<U, std::vector<complex<double>>>   ||
                  std::is_same_v<U, std::vector<complex<float>>>    ||
                  std::is_same_v<U, std::vector<complex<int>>>      ||
                  std::is_same_v<U, std::string>){
        return std::vector<int>{static_cast<int>(inputTensor.size())};
    }
    else if constexpr (std::is_same_v<U, std::vector<std::vector<double>>> ||
                       std::is_same_v<U, std::vector<std::vector<float>>> ||
                       std::is_same_v<U, std::vector<std::vector<int>>> ||
                       std::is_same_v<U,
    std::vector<std::vector<std::complex<double>>>> ||
                       std::is_same_v<U,
    std::vector<std::vector<std::complex<float>>>> ||
                       std::is_same_v<U,
    std::vector<std::vector<std::complex<int>>>>){
        return std::vector<int>{static_cast<int>(inputTensor.size()),
```

```
                                      static_cast<int>(inputTensor[0].size())};
    }
}
```

A.20 svr_sum.hpp

```
template <size_t Axis, typename T>
auto sum(const vector<vector<T>>& inputMatrix){

    // asserting dimensions for now
    static_assert(Axis == 0 || Axis == 1, "Axis must be 0 or 1\n");

    // splitting based on dimensions
    if constexpr (Axis == 0){
        auto returnTensor   {std::vector<T>(inputMatrix[0].size(), 0.00)};
        for(auto row = 0; row < inputMatrix.size(); ++row){
            std::transform(inputMatrix[row].begin(),
                           inputMatrix[row].end(),
                           returnTensor.begin(),
                           returnTensor.begin(),
                           std::plus<T>{});
        }
        return returnTensor;
    }
    else{
        auto returnTensor   {std::vector<std::vector<T>>(inputMatrix.size(),
                                                          std::vector<T>(1, 0.00))};
        for(auto row = 0; row < inputMatrix.size(); ++row){
            auto temp   {std::accumulate(inputMatrix[row].begin(),
                                         inputMatrix[row].end(),
                                         T{0})};
            returnTensor[row][0]    = temp;
        }
        return returnTensor;
    }
}
```

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
```