

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 environment with an acoustic signal, recording the returns (echoes) and estimating task-specific characteristics of the underwater environment. For oceanographic tasks, this means sea-floor topology, movement of marine life and presence of alien bodies. For defense tasks, this means detection and neutralizing 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 ubiquitous, 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 environments, 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

Abstract	iii
Acknowledgements	v
1 Single-sensor signal simulation	7
1.1 Aim	7
1.2 Plots	7
1.3 C++ Code	8
1.4 Octave Code	9
2 Simulate the input to a 4 element array.	11
2.1 Aim	11
2.2 Plots	11
2.3 C++ Code	11
2.4 Octave Code	13
3 Narrowband beamformer	15
3.1 Aim	15
3.2 Plots	15
3.3 Observation	16
3.4 C++ Code	16
3.5 Octave Code	18
4 Simulate beam pattern by shifting theta	21

4.1	Aim	21
4.2	Plots	21
4.3	Observation	21
4.4	C++ Code	22
4.5	Octave Code	23
5	Beam Patterns for Frequencies Different from Design-Frequency	25
5.1	Aim	25
5.2	Plots	25
5.3	Observation	26
5.4	C++ Code	26
5.5	Octave Code	28
6	Effect of SNR on beam pattern	31
6.1	Aim	31
6.2	Plots	31
6.3	Observation	33
6.4	C++ Code	33
6.5	Octave Code	34
A	C++ Function Definitions	37
A.1	before.hpp	37
A.2	hashdefines.hpp	37
A.3	usings.hpp	38
A.4	DataStructureDefinitions.hpp	38
A.5	PrintContainers.hpp	38
A.6	TimerClass.hpp	40
A.7	utils.hpp	41

A.8 svr_WriteToCSV.hpp	42
A.9 svr_repmat.hpp	44
A.10 svr_linspace.hpp	45
A.11 svr_fft.hpp	46
A.12 svr_rand.hpp	52
A.13 svr_operator_star.hpp	54
A.14 svr_operators.hpp	57
A.15 svr_tensor_inits.hpp	58
A.16 svr_sin.hpp	59
A.17 svr_slice.hpp	60
A.18 svr_matrix_operations.hpp	61
A.19 svr_shape.hpp	61
A.20 svr_sum.hpp	62
B Octave Function Definitions	63
B.1 fReadCSV.m	63

List of Figures

List of Tables

List of Abbreviations

SONAR	SOund NAvigation And Ranging
RADAR	RAdio Detection And Ranging
SNR	Signal to Noise Ratio
ULA	Uniform Linear Array

Physical Constants

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

List of Symbols

a	distance	m
P	power	W (J s ⁻¹)
ω	angular frequency	rad

*For Dasher,
Labrador Retriever extraordinaire.*

Introduction

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

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

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

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

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

Background

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

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

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

The assumptions under which we perform the simulations are:

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

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

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

Using the above figure, we can calculate the corresponding delay of each element. This will help us in determining to what degree we would have to delay the element

outputs to obtain all the outputs of elements in co phase. Once the outputs are made in-phase, they are added. For an M element array, the co-phase addition increases the signal power N^2 times and the uncorrelated noise power N times. Thus the SNR is enhanced by N times.

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

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

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

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

Using the fourier transform, we get

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

where

- $\tau_i(\theta) = \frac{d_m \cos(\theta)}{c} F_s$
- d_m the distance between the element considered and the element where the wave-front 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.

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

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

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

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

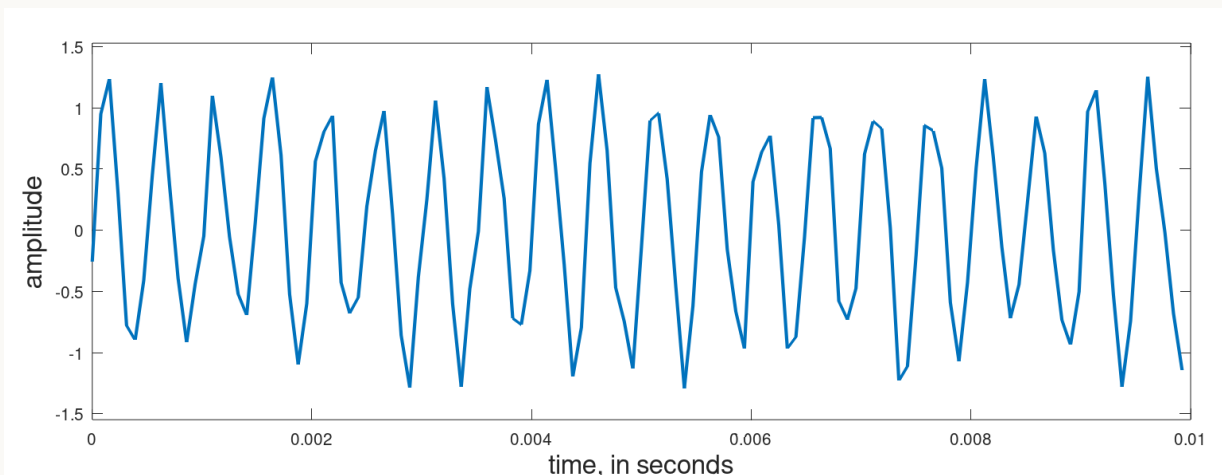


FIGURE 1.1: Time Domain Representation of Signal

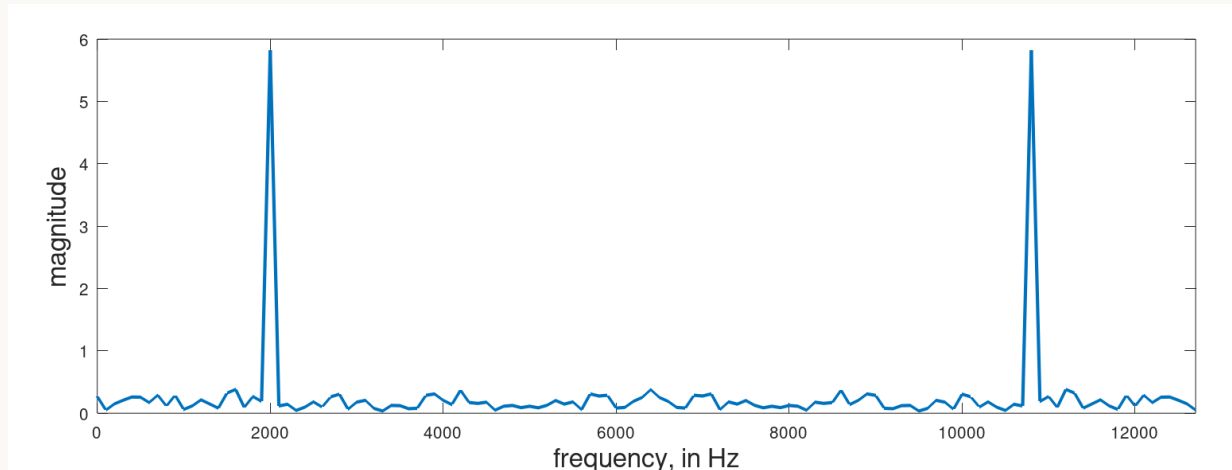


FIGURE 1.2: Magnitude of DFT of input-signal

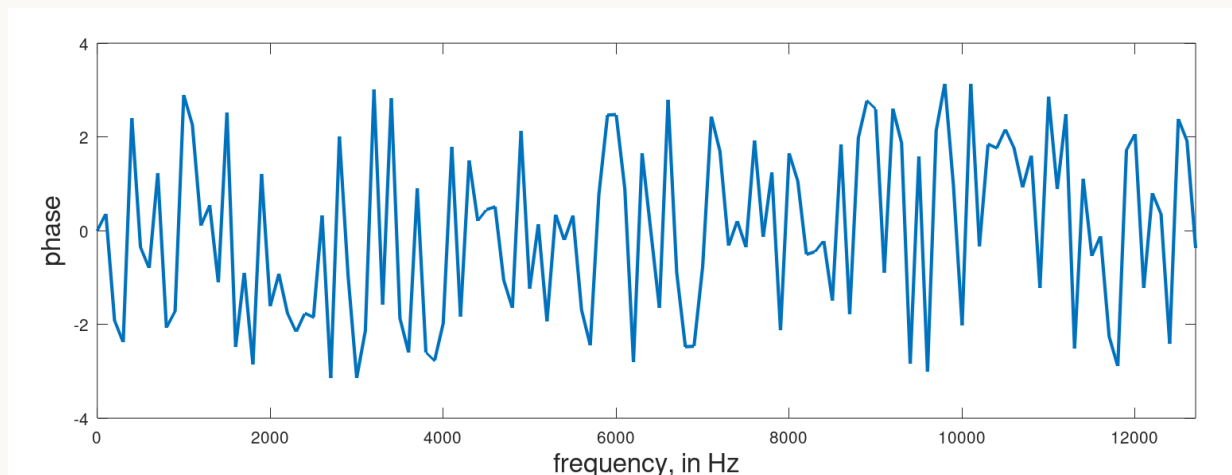


FIGURE 1.3: Phase of DFT of input-signal

1.3 C++ Code

```

1  // =====
2  #include "include/before.hpp"
3  // main-file =====
4  int main(){
5
6      // starting timer
7      auto logfile {string("../csv-files/logfile.csv")};
8      Timer timer(logfile);
9
10     // init-variables
11     auto f {2000}; // frequency of
12     signal // sampling
13     auto Fs {12800}; // corresponding
14     frequency // time-period
15     auto Ts {1/static_cast<double>(Fs)}; // num-samples
16     auto N {128};

```

```

16     auto snr          {10};                                // signal-to-noise
ratio
17     auto snrweight    {std::pow(10, (-1 * snr * 0.05))};    // corresponding
weight
18
19     // building time-array
20     vector<double> t(N,0); t.reserve(N);
21     t = linspace(0.0, static_cast<double>(N-1), static_cast<size_t>(N)) * Ts;
22     fWriteVector(t, "../csv-files/t-Objective1.csv");
23
24     // creating sine-wave
25     auto y = t;
26     std::transform(t.begin(),
27                   t.end(),
28                   y.begin(),
29                   [&](const auto x){return std::sin( 2 * std::numbers::pi * f *
x);});
30
31     // adding noise to the vector
32     auto newmat        {y + snrweight * rand(-1.0, 1.0, y.size())};
33     auto timeaxis      {linspace(static_cast<double>(0),
34                                 static_cast<double>((N-1)*Ts),
35                                 N)};
36     fWriteVector(timeaxis, "../csv-files/timeaxis-Objective1.csv");
37     fWriteVector(newmat,   "../csv-files/newmat-Objective1.csv");
38
39     // Taking the fourier transform
40     auto nfft          {N};
41     auto fend          {static_cast<double>((nfft-1) * Fs) /
static_cast<double>(nfft)};
42     auto waxis         {linspace(static_cast<double>(0),
43                                 static_cast<double>(fend),
44                                 nfft)};
45     auto Fourier       {fft(newmat)};
46     fWriteVector(waxis,   "../csv-files/waxis-Objective1.csv");
47     fWriteVector(Fourier,  "../csv-files/Fourier-Objective1.csv");
48
49     // return
50     return(0);
51
52 }

```

1.4 Octave Code

```

1  %% Basic Setup
2  clc; clear all; close all;
3  addpath("../include/")
4
5  %% Loading the files
6  timeaxis    = csvread("../csv-files/timeaxis-Objective1.csv");
7  newmat      = csvread("../csv-files/newmat-Objective1.csv");
8  waxis       = csvread("../csv-files/waxis-Objective1.csv");
9  newmatfft   = fReadCSV("../csv-files/Fourier-Objective1.csv");
10

```

```
11 %% Plotting
12 plotwidth = 1515;
13 plotheight = 500;
14
15 figure(1)
16 set(gcf, 'Position', [0 0 plotwidth plotheight]); % [left bottom width height]
17 plot(timeaxis, newmat, "LineWidth", 2);
18 xlabel("time, in seconds", "fontsize", 16);
19 ylabel("amplitude", "fontsize", 16);
20 ylim([1.2 * min(newmat), 1.2 * max(newmat)]);
21 saveas(gcf, "../Figures/y-objective1.png");
22
23 figure(2);
24 set(gcf, 'Position', [0 0 plotwidth plotheight]); % [left bottom width height]
25 plot(waxis, abs(newmatfft), "LineWidth", 2);
26 xlabel("frequency, in Hz", "fontsize", 16);
27 ylabel("magnitude", "fontsize", 16);
28 xlim([min(waxis), max(waxis)]);
29 saveas(gcf, "../Figures/abs-yfft-objective1.png");
30
31 figure(3);
32 set(gcf, 'Position', [0 0 plotwidth plotheight]); % [left bottom width height]
33 plot(waxis, angle(newmatfft), "LineWidth", 2);
34 xlabel("frequency, in Hz", "fontsize", 16);
35 ylabel("phase", "fontsize", 16);
36 xlim([min(waxis), max(waxis)]);
37 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

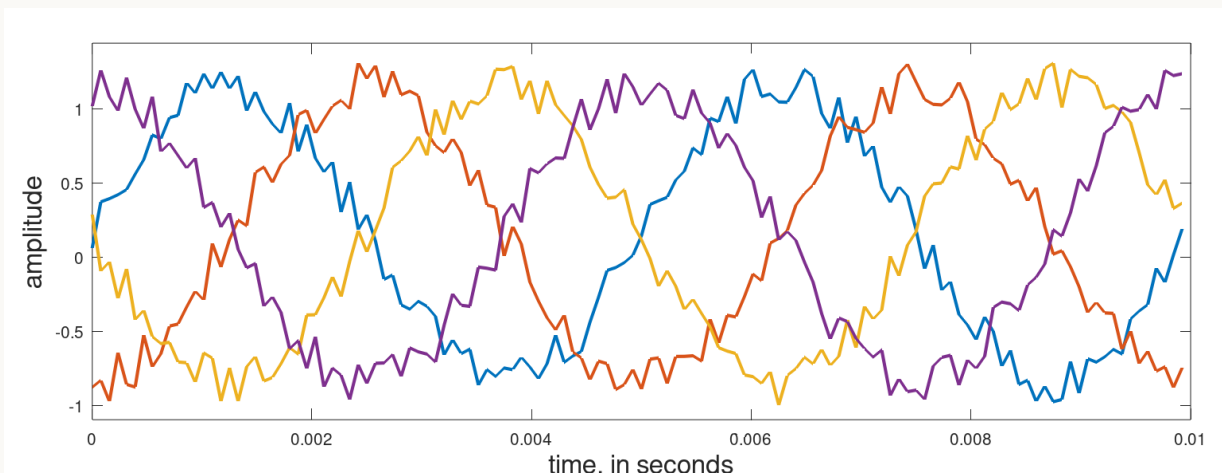


FIGURE 2.1: Time Domain Representation of Signal

2.3 C++ Code

```

1 // =====
2 #include "include/before.hpp"
3 // main-file =====
4 int main(){
5
6     // starting timer
7     auto logfile {string("../csv-files/logfile.csv")};
8     Timer timer(logfile);
9
10    // init-variables
11    auto f {200}; // frequency
12    of signal
13    auto Fs {12800}; // sampling
14    frequency
15    auto Ts {1/static_cast<double>(Fs)}; //
16    corresponding time-period
17    auto N {128}; //
18    num-samples
19
20    auto m {4};
21    auto angleofarrival {60};
22    auto speedofsound {1500};
23    auto lambda
24    {static_cast<double>(speedofsound)/static_cast<double>(f)};
25    auto x {lambda/2};
26    auto d {x * std::cos(angleofarrival * std::numbers::pi / 180)
27    / speedofsound};
28
29    auto snr {10}; //
30    signal-to-noise ratio
31    auto snrweight {std::pow(10, (-1 * snr * 0.05))}; //
32    corresponding weight
33
34    // building time-array
35    vector<double> t(N,0); t.reserve(N);
36    t = linspace(0.0,
37    static_cast<double>(N-1),
38    static_cast<size_t>(N)) * Ts;
39    fWriteVector(t, "../csv-files/timeaxis-Objective2.csv");
40
41    // building matrix
42    auto matrix = Zeros({m, N});
43
44    // creating sine-wave
45    auto y = sin(2 * std::numbers::pi * f * t);
46    fWriteVector(y, "../csv-files/y-Objective2.csv");
47
48    // building the matrix
49    for(int i = 0; i<m; ++i)
50        matrix[i] = sin(2 * std::numbers::pi * f * (t - i * d));
51    fWriteMatrix(matrix, "../csv-files/matrix-Objective2.csv");
52
53    // Adding noise to the matrix
54    vector<vector<double>> additivenoise = snrweight * rand(0.0, 1.0, {m, N});
55    fWriteMatrix(additivenoise, "../csv-files/additivenoise-Objective2.csv");
56
57    auto newmat {matrix + additivenoise};

```

```
50     fWriteMatrix(newmat, "../csv-files/newmat-Objective2.csv");
51
52     // return
53     return(0);
54
55 }
```

2.4 Octave Code

```
1  %% Basic Setup
2  clc; clear all; close all;
3  addpath("../include/")
4
5  %% Loading the files
6  timeaxis = csvread("../csv-files/timeaxis-Objective2.csv");
7  newmat    = csvread("../csv-files/newmat-Objective2.csv");
8
9  %% Plotting
10 plotwidth = 1515;
11 plotheight = 500;
12
13 figure(1)
14 set(gcf, 'Position', [0 0 plotwidth plotheight]); % [left bottom width height]
15 plot(timeaxis, newmat, "LineWidth", 2);
16 xlabel("time, in seconds", "fontsize", 16);
17 ylabel("amplitude", "fontsize", 16);
18 ylim([1.1 * min(newmat(:)), 1.1 * max(newmat(:))]);
19 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

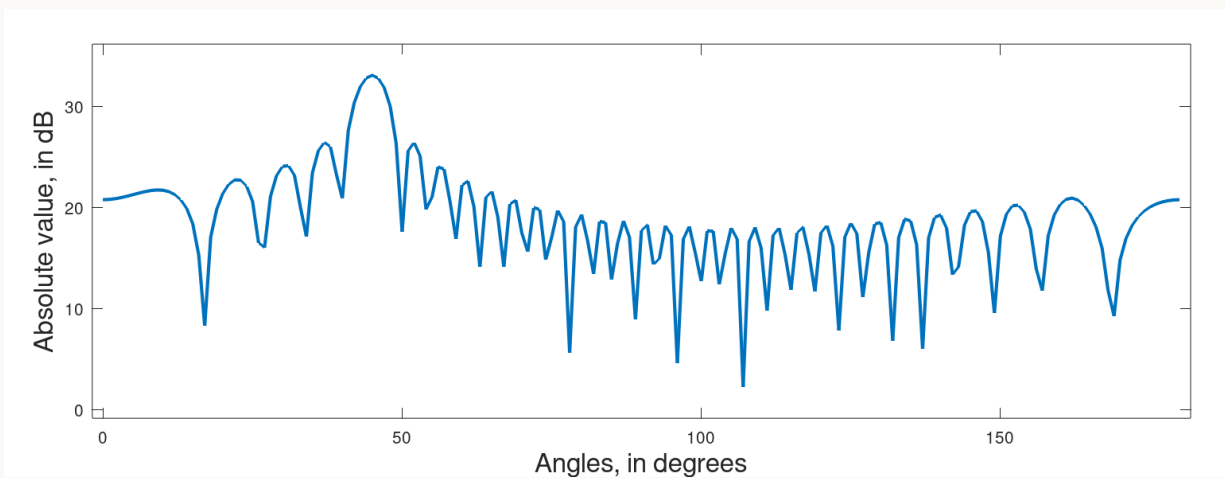


FIGURE 3.1: Beam-pattern for beamformer at angle 45

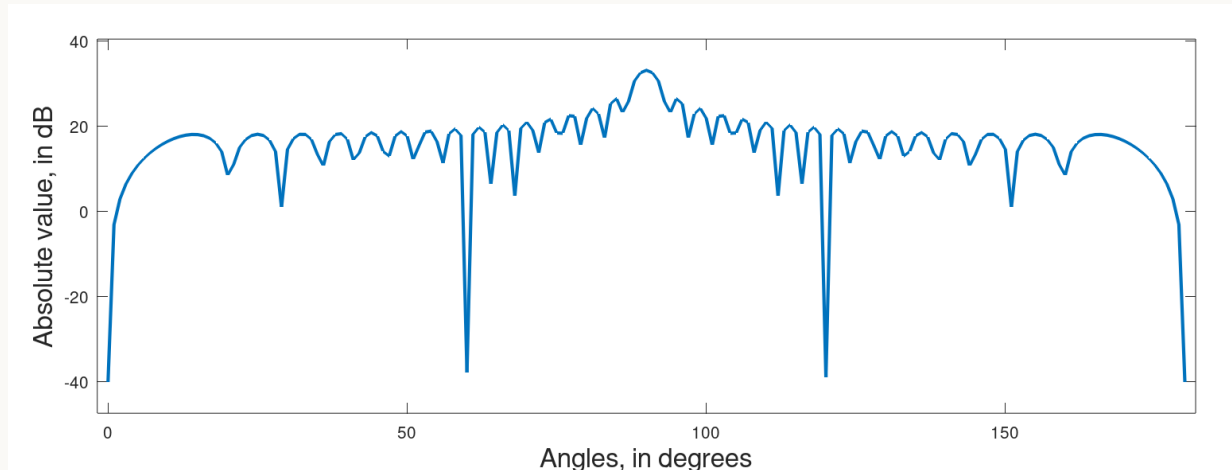


FIGURE 3.2: Beam-pattern for beamformer at angle 90

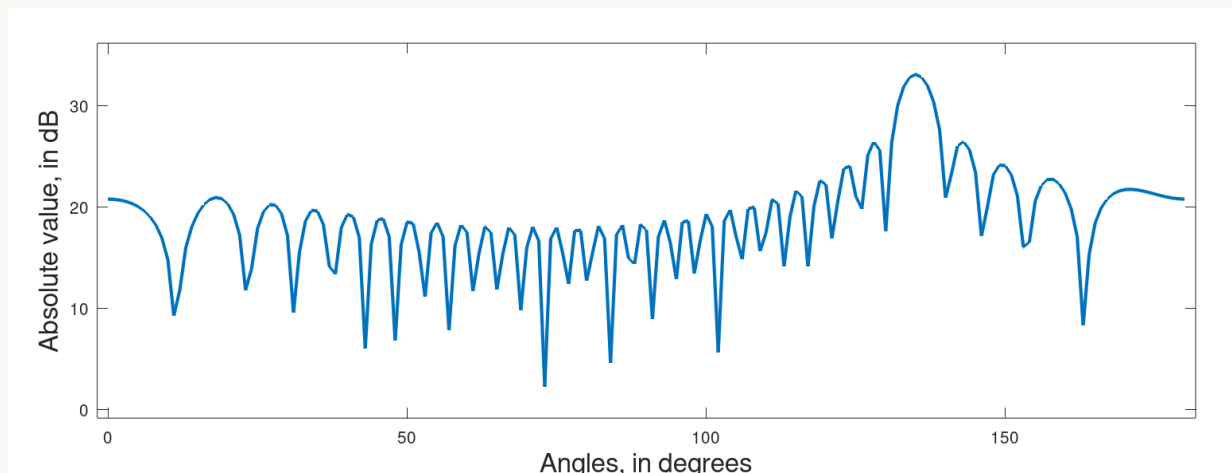


FIGURE 3.3: Beam-pattern for beamformer at angle 135

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

```

1 // =====
2 #include "include/before.hpp"
3 // main-file =====
4 int main(){
5
6     // starting timer
7     auto logfile {string("../csv-files/logfile-Objective3.csv")};
8     Timer timer(logfile);
9
10    // init-variables

```

```

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

15
16     auto m               {32};
17     auto angleofarrival   {135};
18     auto speedofsound     {1500};
19     auto lambda
    {static_cast<double>(speedofsound)/static_cast<double>(f)};
20     auto x               {lambda/2.00};
21     auto d               {x * std::cos(static_cast<double>(angleofarrival) *
    std::numbers::pi / 180) / speedofsound};

22
23     auto snr              {100};                                    //
    signal-to-noise ratio
24     auto snrweight        {std::pow(10, (-1 * snr * 0.05))};      //
    corresponding weight

25
26
27     // building time-array
28     vector<double>t = linspace(0.0,
                                static_cast<double>(N-1) * Ts,
                                static_cast<size_t>(N));
29
30     fWriteVector(t, "../csv-files/t-Objective3.csv");
31
32
33     // building matrix
34     auto matrix = Zeros({m, N});
35
36     // creating sine-wave
37     auto y              {sin(2 * std::numbers::pi * f * t)};
38     fWriteVector(y, "../csv-files/y-Objective3.csv");
39
40     // building the matrix
41     for(int sensorindex = 0; sensorindex < m; ++sensorindex)
42         matrix[sensorindex] = sin(2.00 * std::numbers::pi * f * (t - sensorindex *
    d));
43     fWriteMatrix(matrix, "../csv-files/matrix-Objective3.csv");
44
45
46     // Adding noise to the matrix
47     auto newmat          {matrix + snrweight * rand(0.0, 1.0, {m, N})};
48     fWriteMatrix(newmat, "../csv-files/newmat-Objective3.csv");
49
50
51     // Taking the fourier-transform
52     auto nfft            {N};
53     auto fend            {static_cast<double>(nfft - 1) * static_cast<double>(Fs) /
    static_cast<double>(nfft)};
54     auto waxis           {linspace(0, fend, nfft)};
55     auto Fourier         {fft(newmat, nfft)};
56     fWriteMatrix(Fourier, "../csv-files/Fourier-Objective3.csv");
57
58

```

```

59 // choosing the frequency row
60 int index = std::floor(static_cast<double>(f)/
61 (static_cast<double>(Fs)/static_cast<double>(nfft)));
62 auto fmat {slice(Fourier, {-1, -2, index, index})};
63
64 // Bringing the delay in frequency region
65 auto anglematrix {vector<double>(181)};
66 auto delaycolumn {vector<complex<double>>(m)};
67
68 // moving through angle
69 for(int testangle = 0; testangle<181; ++testangle){
70
71     double testd {x * cosd(testangle) / speedofsound};
72
73     for(int currsensor = 0; currsensor < m; ++currsensor){
74         delaycolumn[currsensor] = \
75             std::exp( 1 * std::complex<double>{0, 1} * 2 * std::numbers::pi * f
76 * currsensor * testd);
77     }
78
79     // calculating inner-product
80     auto innerproduct_value {delaycolumn[0] * fmat[0][0]};
81     for(int i = 1; i<delaycolumn.size(); ++i)
82         innerproduct_value += delaycolumn[i] * fmat[i][0];
83
84     // storing to the angle-matrix
85     anglematrix[testangle] = std::abs(innerproduct_value);
86 }
87 fWriteVector(anglematrix, "../csv-files/anglematrix-Objective3.csv");
88
89 // creating angle-axis
90 auto angleaxis {linspace(0, 180, 181)};
91 fWriteVector(angleaxis, "../csv-files/angleaxis-Objective3.csv");
92
93 // return
94 return(0);
95 }

```

3.5 Octave Code

```

1  %% Basic Setup
2  clc; clear all;
3
4  %% Loading the files
5  timearray = csvread("../csv-files/t-Objective3.csv");
6  yarray = csvread("../csv-files/y-Objective3.csv");
7  matrix = csvread("../csv-files/matrix-Objective3.csv");
8
9  ULAMatrix = csvread("../csv-files/newmat-Objective3.csv");
10 FourierMatrix = fReadCSV("../csv-files/Fourier-Objective3.csv");
11
12 angleaxis = csvread("../csv-files/angleaxis-Objective3.csv");

```



```
13 anglematrix      = csvread("../csv-files/anglematrix-Objective3.csv");
14 anglematrixdB     = 10*log10(anglematrix);
15
16 %% Plotting the signals
17 plotwidth        = 1515;
18 plotheight       = 500;
19
20 figure(1);
21 set(gcf, 'Position', [0 0 plotwidth plotheight]); % [left bottom width height]
22 plot(angleaxis, anglematrixdB, "LineWidth", 2);
23 xlabel("Angles, in degrees",      "fontsize", 16);
24 ylabel("Absolute value, in dB",   "fontsize", 16);
25 ylim([min(anglematrixdB) - 1e-1 * range(anglematrixdB),
26       max(anglematrixdB) + 1e-1 * range(anglematrixdB)]);
27 xlim([min(angleaxis) - 1e-2 * range(angleaxis),
28       max(angleaxis) + 1e-2 * range(angleaxis)]);
29 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

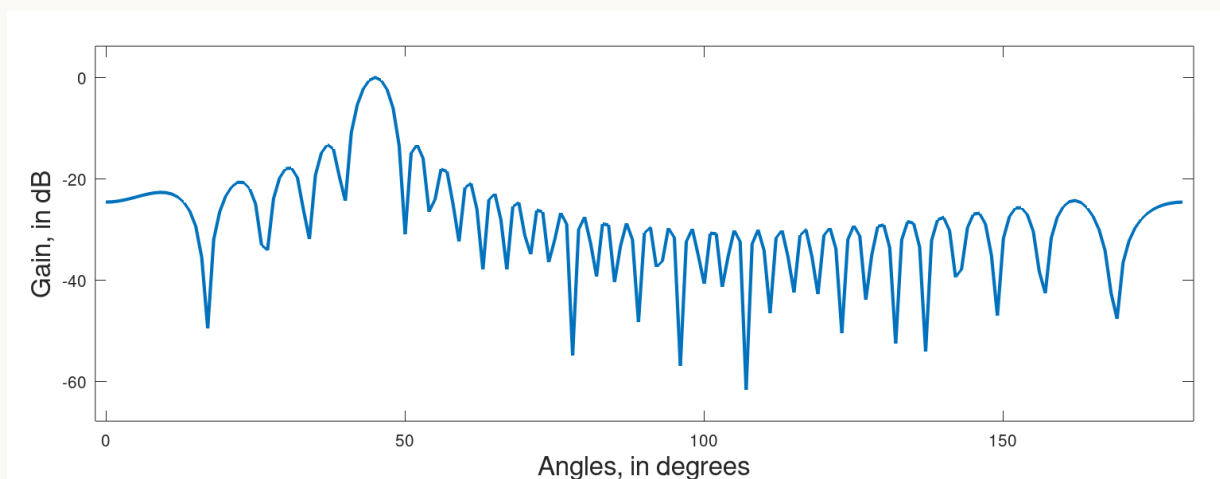


FIGURE 4.1: Beam-pattern for beamformer at angle 45

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.

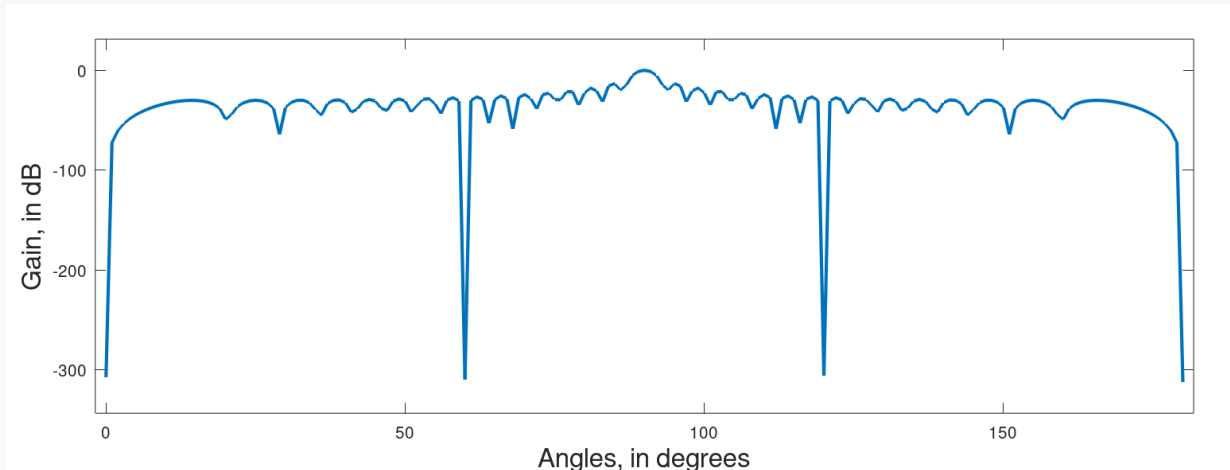


FIGURE 4.2: Beam-pattern for beamformer at angle 90

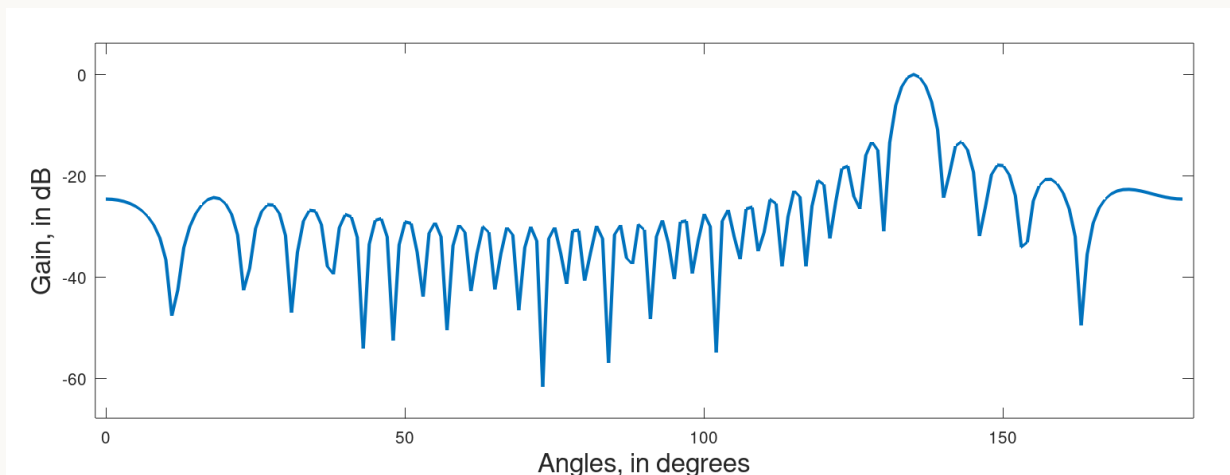


FIGURE 4.3: Beam-pattern for beamformer at angle 135

4.4 C++ Code

```

1  // =====
2  #include "include/before.hpp"
3  // main-file =====
4  int main(){
5
6      // starting timer
7      auto logfile {string("logfile.csv")};
8      Timer timer(logfile);
9
10     // init-variables
11     auto f {2000};
12     auto m {static_cast<double>(32)};
13     auto angle {135};
14     auto c {1500};
15     auto lambda {static_cast<double>(c)/static_cast<double>(f)};
16     auto x {lambda/2};
17     auto d {x * cosd(angle)/c};
18
19     // bringing about the natural delay

```

```

20     auto matrix          {vector<complex<double>>(m, complex<double>(0))};
21     for(auto sensorindex = 0; sensorindex < m; ++sensorindex){
22         matrix[sensorindex] = \
23             (1.00/static_cast<double>(m)) * \
24             std::exp(-1.00 * 1i * 2.00 * \
25                 std::numbers::pi * f * (sensorindex) * d);
26     }
27
28     // bringing the delay in frequency region
29     auto delaycolumn      = vector<complex<double>>(m, complex<double>(0));
30     auto anglematrix      = vector<double>(181, 0);
31
32     // calculating
33     for(int testangle = 0; testangle < 181; ++testangle){
34
35         auto testd {x * cosd(testangle)/c};
36
37         for(int sensorindex = 0; sensorindex < m; ++sensorindex)
38             delaycolumn[sensorindex] = \
39                 std::exp(1.00 * 1i * 2.00 * \
40                     std::numbers::pi * f * sensorindex * testd);
41
42         // performing inner-product
43         anglematrix[testangle] = \
44             std::abs(std::inner_product(matrix.begin(),
45                                         matrix.end(),
46                                         delaycolumn.begin(),
47                                         complex<double>{0}));
48
49     }
50
51     // producing angle axis
52     auto angleaxis {linspace(0, 180, 181)};
53
54     // saving the tensors
55     fWriteVector(angleaxis,      "../Figures/angleaxis-Objective4.csv");
56     fWriteVector(anglematrix,    "../Figures/anglematrix-Objective4.csv");
57
58     // return
59     return(0);
60
61 }

```

4.5 Octave Code

```

1  %% Basic Setup
2  clc; clear all; close all;
3
4  %% Loading the files
5  angleaxis      = csvread("../Figures/angleaxis-Objective4.csv");
6  anglematrix    = csvread("../Figures/anglematrix-Objective4.csv");
7  anglematrixinDB = 20 * log10(anglematrix);
8
9  %% Plotting the signals

```

```
10 plotwidth    = 1515;
11 plotheight   = 500;
12
13 figure(1);
14 set(gcf, 'Position', [0 0 plotwidth plotheight]); % [left bottom width height]
15 plot(angleaxis, anglematrixinDB, "LineWidth", 2);
16 xlabel("Angles, in degrees", "fontsize", 16);
17 ylabel("Gain, in dB", "fontsize", 16);
18 ylim([min(anglematrixinDB) - 1e-1 * range(anglematrixinDB),
19      max(anglematrixinDB) + 1e-1 * range(anglematrixinDB)]);
20 xlim([min(angleaxis) - 1e-2 * range(angleaxis),
21      max(angleaxis) + 1e-2 * range(angleaxis)]);
22 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

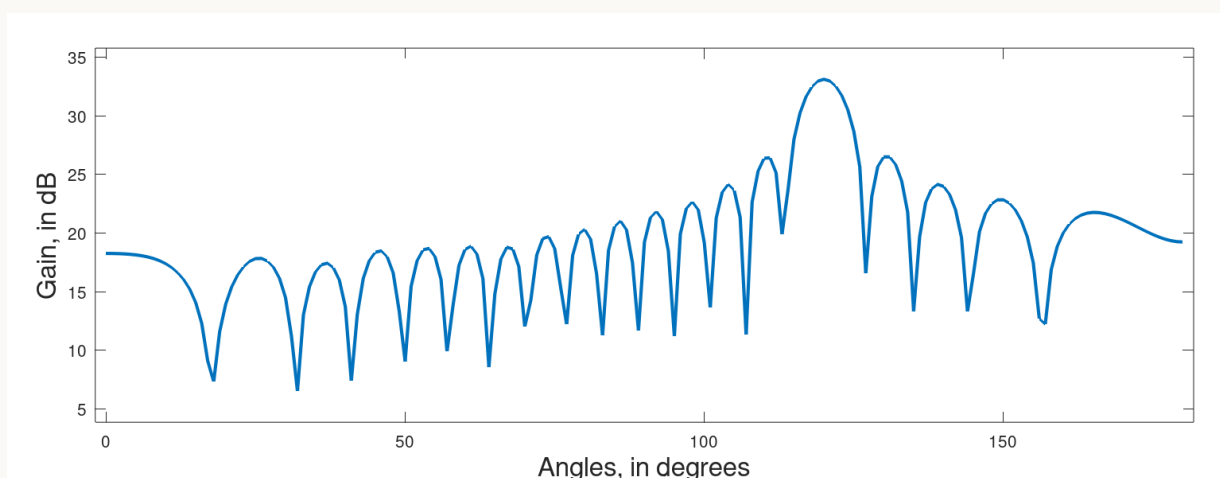


FIGURE 5.1: Beamformed for 1200Hz

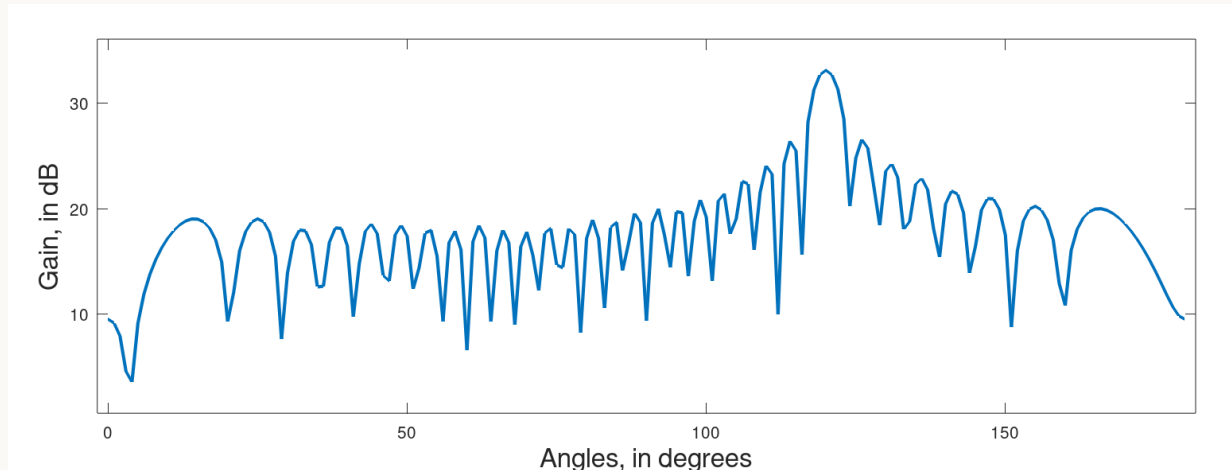


FIGURE 5.2: Beamformed for 2000Hz

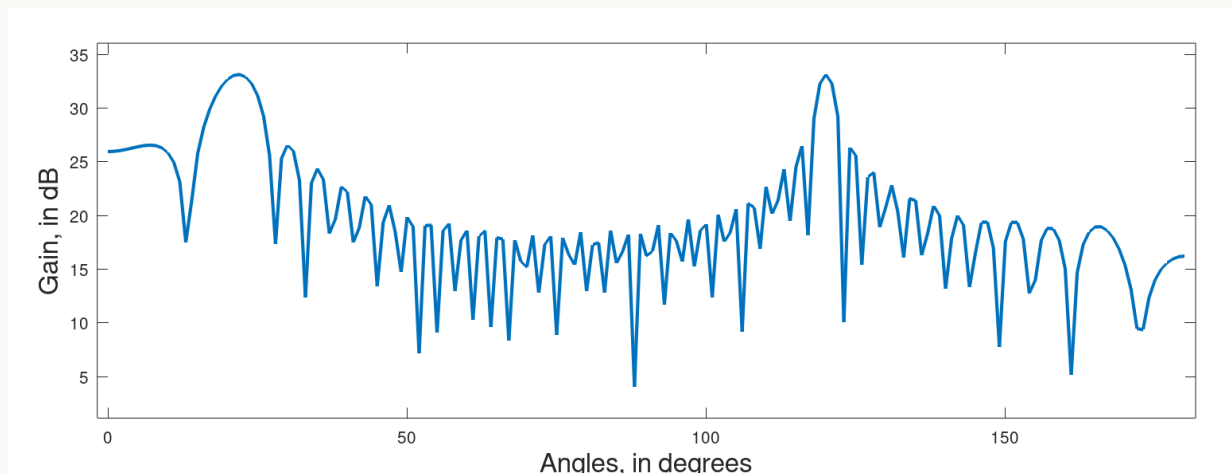


FIGURE 5.3: Beamformed for 2800Hz

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

```

1 // =====
2 #include "include/before.hpp"
3 // main-file =====
4 int main(){
5
6     // starting timer

```



```

7   auto logfile    {string("../csv-files/logfile.csv")};
8   Timer timer(logfile);
9
10  // init-variables
11  auto f           {2800};
12  auto f_1         {2000};
13  auto Fs          {12800};
14  auto Ts          {1.00/static_cast<double>(Fs)};
15  auto N           {128};
16
17  auto m           {32};
18  auto angle       {120};
19  auto c           {1500};
20  auto lambda      {static_cast<double>(c)/static_cast<double>(f_1)};
21  auto x           {lambda/2.00};
22  auto d           {static_cast<double>(x * cosd(angle)/c)};
23
24  auto snr         {10};
25  auto snrweight   {static_cast<double>(std::pow(10, -1 * snr * 0.05))};
26
27
28  // creating tensors
29  auto t           {linspace(static_cast<double>(0),
30                           static_cast<double>(N-1) * Ts,
31                           N)};
32  auto matrix      {Zeros({m, N})};
33
34
35  // bringing about natural delay
36  for(int sensorindex = 0; sensorindex < m ; ++sensorindex)
37      matrix[sensorindex] = \
38          sin( 2.00 * std::numbers::pi * f * (t - sensorindex * d));
39  fWriteVector(t,      "../csv-files/timearray-Objective5.csv");
40  fWriteMatrix(matrix, "../csv-files/matrix-Objective5.csv");
41
42
43  // adding the noise
44  auto newmat = matrix + snrweight * rand(0.00, 1.00, {m, N});
45  fWriteMatrix(newmat, "../csv-files/newmat-Objective5.csv");
46
47
48  // taking the fourier transform
49  auto nfft       {N};
50  auto fend       {static_cast<double>((nfft-1) * Fs) /
51  static_cast<double>(nfft)};
52  auto waxis      {linspace(0, fend, nfft)};
53  auto Fourier    {fft(newmat, nfft)};
54  fWriteVector(waxis,  "../csv-files/waxis.csv");
55  fWriteMatrix(Fourier, "../csv-files/Fourier.csv");
56
57  // choosing the frequency row
58  int index       {static_cast<int>(std::floor(static_cast<double>(f)/
59  (static_cast<double>(Fs)/static_cast<double>(nfft))))};
60  auto fmat       {slice(Fourier, {-1, -2, index, index})};
61
62  // bringing the delay in frequency region
63  auto anglematrix {vector<double>(181)};
64  auto delaycolumn {vector<complex<double>>(m, 0)};

```

```

63
64 // building
65 for(int testangle = 0; testangle < 181; ++testangle){
66
67     auto testd {x * cosd(testangle)/c};
68
69     for(int sensorindex = 0; sensorindex < m; ++sensorindex)
70         delaycolumn[sensorindex] = std::exp(1 * 1i * 2 * std::numbers::pi *
71 f * sensorindex * testd);
72
73     anglematrix[testangle] = \
74         std::abs(std::inner_product(fmat.begin(), fmat.end(),
75 delaycolumn.begin(),
76 complex<double>{0},
77 std::plus<complex<double>>(),
78 [] (vector<complex<double>> a,
79 complex<double> b){
80         return a[0]*b;
81     }));
82
83 // building axes
84 auto angleaxis = linspace(0, 180, 181);
85
86 // saving
87 fWriteVector(angleaxis,    "../csv-files/angleaxis-Objective-5.csv");
88 fWriteVector(anglematrix,  "../csv-files/anglematrix-Objective-5.csv");
89
90 // return
91 return(0);
92 }

```

5.5 Octave Code

```

1  %% Basic Setup
2  clc; clear all;
3
4  %% Loading the files
5  angleaxis = csvread("../csv-files/angleaxis-Objective-5.csv");
6  anglegains = csvread("../csv-files/anglematrix-Objective-5.csv");
7  anglegainsindB = 10 * log10(anglegains);
8
9  %% Plotting the signals
10 plotwidth = 1515;
11 plotheight = 500;
12
13 figure(1);
14 set(gcf, 'Position', [0 0 plotwidth plotheight]); % [left bottom width height]
15 plot(angleaxis, anglegainsindB, "LineWidth", 2);
16 xlabel("Angles, in degrees", "fontsize", 16);
17 ylabel("Gain, in dB", "fontsize", 16);
18 ylim([min(anglegainsindB) - 1e-1 * range(anglegainsindB), max(anglegainsindB) +
1e-1 * range(anglegainsindB)]);

```

```
19 xlim([min(angleaxis) - 1e-2 * range(angleaxis), max(angleaxis) + 1e-2 *  
    range(angleaxis)]);  
20 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

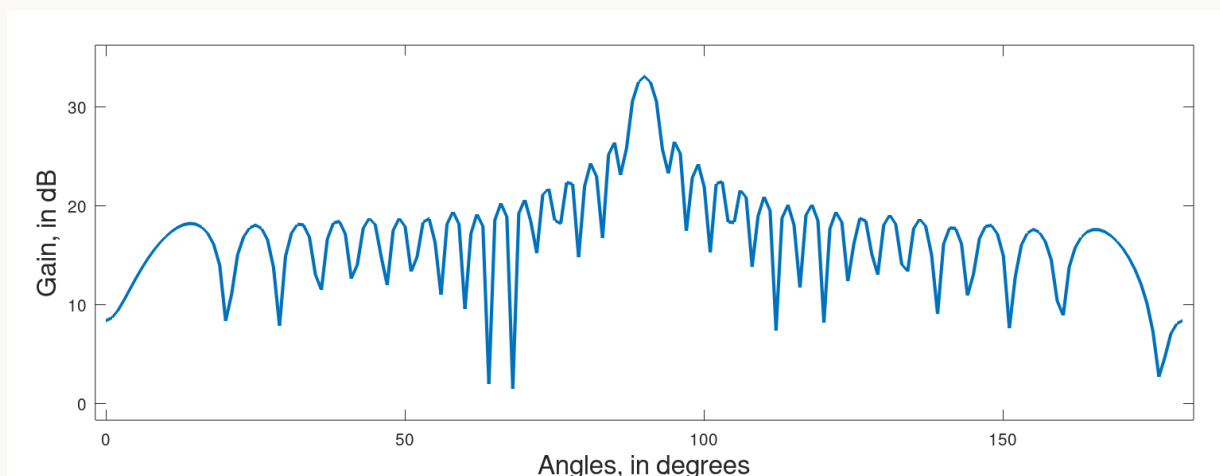


FIGURE 6.1: Beamforming for SNR = 10

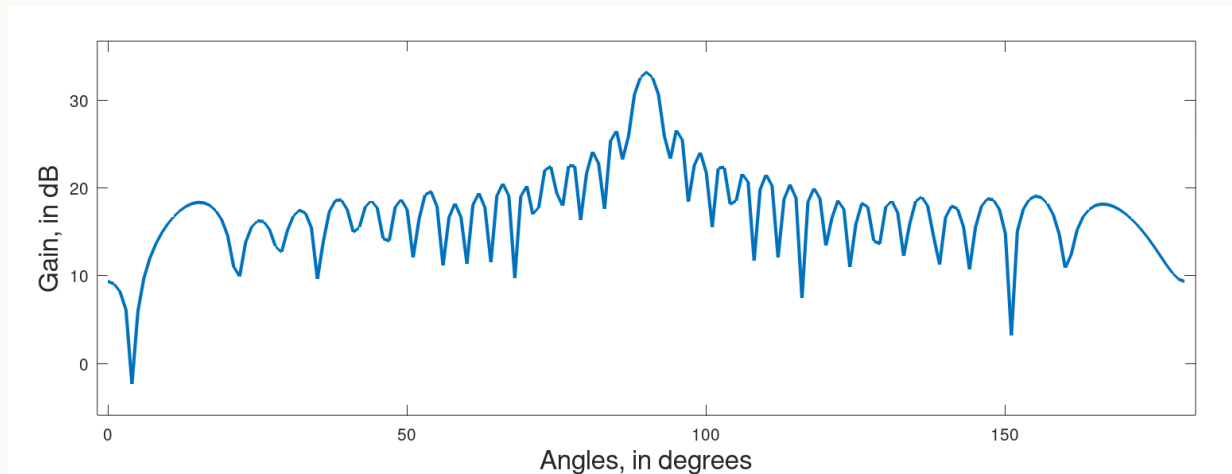


FIGURE 6.2: Beamforming for SNR = -1

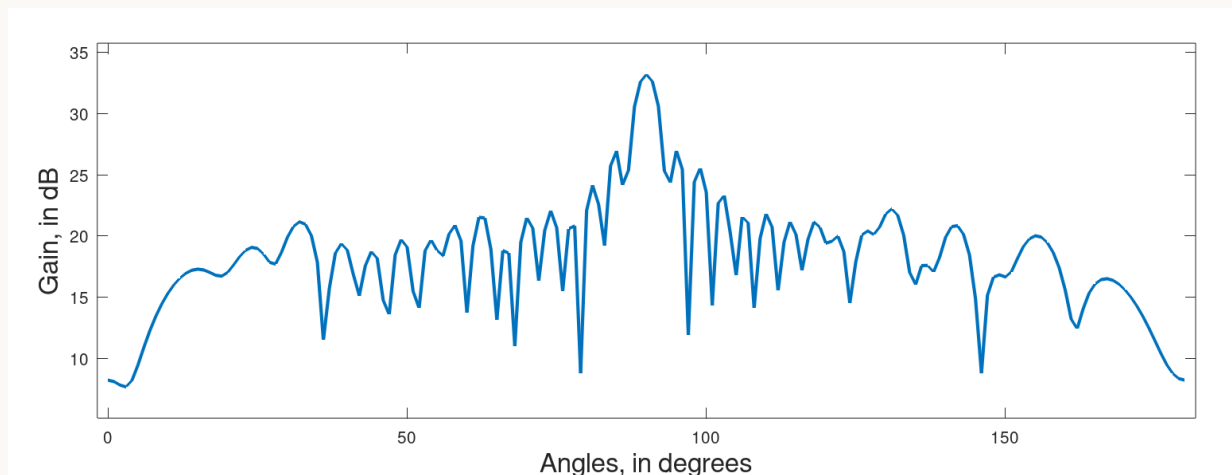


FIGURE 6.3: Beamforming for SNR = -10

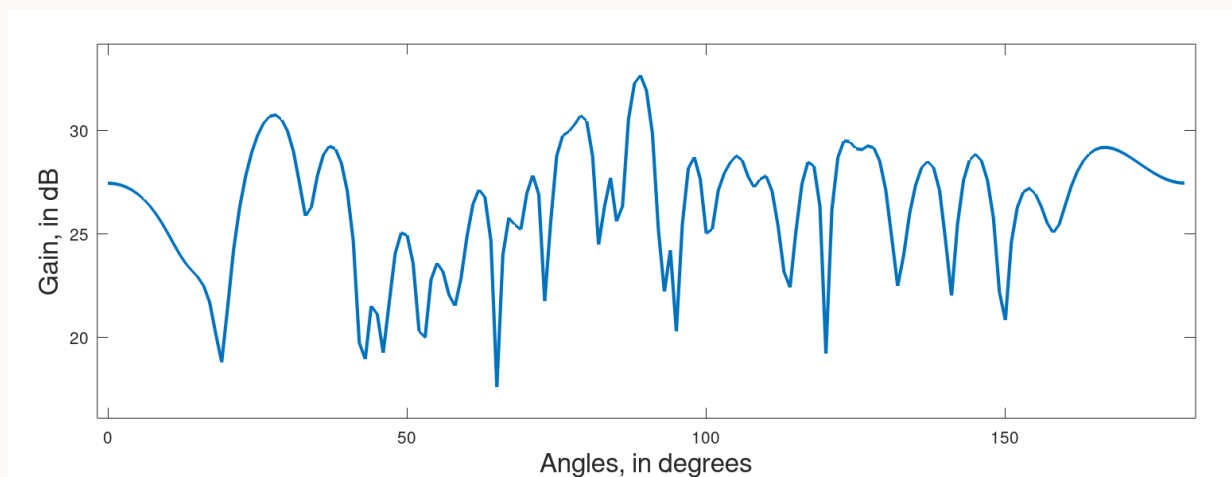


FIGURE 6.4: Beamforming for SNR = -30

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

```

1 // =====
2 #include "include/before.hpp"
3 // main-file =====
4 int main(){
5
6     // starting timer
7     auto logfile {string("../csv-files/logfile-Objective6.csv")};
8     Timer timer(logfile);
9
10    // init-variables
11    auto f          {2800};
12    auto Fs         {12800};
13    auto Ts         {1.00/static_cast<double>(Fs)};
14    auto N          {128};
15
16    auto m          {32};
17    auto angle      {90};
18    auto c          {1500};
19    auto lambda     {static_cast<double>(c)/static_cast<double>(f)};
20    auto x          {lambda/2.00};
21    auto d          {static_cast<double>(x * cosd(angle)/c)};
22
23    auto snr        {30};
24    auto snrweight  {static_cast<double>(std::pow(10, -1 * snr * 0.05))};
25
26
27    // simulating signals
28    auto t          {linspace(static_cast<double>(0),
29                               static_cast<double>(N-1) * Ts,
30                               N)};
31    auto matrix     {Zeros({m, N})};
32
33
34    // bringing about natural delay
35    for(int sensorindex = 0; sensorindex < m ; ++sensorindex)
36        matrix[sensorindex] = \
37            sin( 2.00 * std::numbers::pi * f * (t - sensorindex * d));
38    fWriteVector(t,          "../csv-files/timearray-Objective6.csv");
39    fWriteMatrix(matrix,     "../csv-files/matrix-Objective6.csv");
40
41
42    // adding the noise
43    auto newmat = matrix + snrweight * rand(0.00, 1.00, {m, N});
44    fWriteMatrix(newmat,     "../csv-files/newmat-Objective6.csv");
45

```

```

46
47 // taking the fourier transform
48 auto nfft      {N};
49 auto fend      {static_cast<double>((nfft-1) * Fs) /
static_cast<double>(nfft)};
50 auto waxis      {linspace(0, fend, nfft)};
51 auto Fourier    {fft(newmat, nfft)};
52 fWriteVector(waxis,      "../csv-files/waxis.csv");
53 fWriteMatrix(Fourier,    "../csv-files/Fourier.csv");
54
55 // choosing the frequency row
56 int index      {static_cast<int>(std::floor(static_cast<double>(f)/
(static_cast<double>(Fs)/static_cast<double>(nfft)))));
57 auto fmat      {slice(Fourier, {-1, -2, index, index})};
58
59 // bringing the delay in frequency region
60 auto anglematrix {vector<double>(181)};
61 auto delaycolumn {vector<complex<double>>(m, 0)};
62
63 // building
64 for(int testangle = 0; testangle < 181; ++testangle){
65
66     auto testd {x * cosd(testangle)/c};
67
68     for(int sensorindex = 0; sensorindex < m; ++sensorindex)
69         delaycolumn[sensorindex] = std::exp(1 * 1i * 2 * std::numbers::pi *
f * sensorindex * testd);
70
71     anglematrix[testangle] = \
72         std::abs(std::inner_product(fmat.begin(), fmat.end(),
73                                     delaycolumn.begin(),
74                                     complex<double>{0},
75                                     std::plus<complex<double>>(),
76                                     [] (vector<complex<double>> a,
complex<double> b){
77
78                                     return a[0]*b;
79                                     }));
80
81 // building axes
82 auto angleaxis = linspace(0, 180, 181);
83
84 // saving
85 fWriteVector(angleaxis,      "../csv-files/angleaxis-Objective-6.csv");
86 fWriteVector(anglematrix,    "../csv-files/anglematrix-Objective-6.csv");
87
88 // return
89 return(0);
90
91 }

```

6.5 Octave Code

```

1 %% Basic Setup

```



```
2  clc; clear all;
3
4  %% Loading the files
5  angleaxis      = csvread("../csv-files/angleaxis-Objective-6.csv");
6  anglegains     = csvread("../csv-files/anglematrix-Objective-6.csv");
7  anglegainsindB = 10 * log10(anglegains);
8
9  %% Plotting the signals
10 plotwidth      = 1515;
11 plotheight     = 500;
12
13 figure(1);
14 set(gcf, 'Position', [0 0 plotwidth plotheight]); % [left bottom width height]
15 plot(angleaxis, anglegainsindB, "LineWidth", 2);
16 xlabel("Angles, in degrees", "fontsize", 16);
17 ylabel("Gain, in dB", "fontsize", 16);
18 ylim([min(anglegainsindB) - 1e-1 * range(anglegainsindB), max(anglegainsindB) +
19       1e-1 * range(anglegainsindB)]);
19 xlim([min(angleaxis) - 1e-2 * range(angleaxis), max(angleaxis) + 1e-2 *
20       range(angleaxis)]);
21 saveas(gcf, "../Figures/anglegainsindB-Objective6.png");
```


Appendix A

C++ Function Definitions

A.1 before.hpp

```

1  // including header-files
2  #include <algorithm>
3  #include <complex>
4  #include <bitset>
5  #include <climits>
6  #include <cstdint>
7  #include <iostream>
8  #include <limits>
9  #include <map>
10 #include <new>
11 #include <stdlib.h>
12 #include <unordered_map>
13 #include <vector>
14 #include <set>
15 #include <numeric>
16 #include <fstream>
17 #include <numbers>
18 #include <cmath>
19 #include <random>
20
21 // custom definitions
22 #include "hashdefines.hpp"
23 #include "usings.hpp"
24 #include "DataStructureDefinitions.hpp"
25 #include "PrintContainers.hpp"
26 #include "TimerClass.hpp"
27 #include "utils.hpp"

```

A.2 hashdefines.hpp

```

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

```

A.3 usings.hpp

```
1  // borrowing from namespace std
2  using std::cout;
3  using std::complex;
4  using std::endl;
5  using std::vector;
6  using std::string;
7  using std::unordered_map;
8  using std::map;
9  using std::format;
10 using std::deque;
11 using std::pair;
12 using std::min;
13 using std::max;
14 using namespace std::complex_literals;
```

A.4 DataStructureDefinitions.hpp

```
1  struct TreeNode {
2      int val;
3      TreeNode *left;
4      TreeNode *right;
5      TreeNode() : val(0), left(nullptr), right(nullptr) {}
6      TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
7      TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
8          right(right) {}
9  };
10
11 struct ListNode {
12     int val;
13     ListNode *next;
14     ListNode() : val(0), next(nullptr) {}
15     ListNode(int x) : val(x), next(nullptr) {}
16     ListNode(int x, ListNode *next) : val(x), next(next) {}
17 };
```

A.5 PrintContainers.hpp

```
1  // vector printing function
2  template<typename T>
3  void fPrintVector(vector<T> input){
4      for(auto x: input) cout << x << ", ";
5      cout << endl;
6  }
7
8  template<typename T>
```

```

9  void fpv(vector<T> input){
10     for(auto x: input) cout << x << ",";
11     cout << endl;
12 }
13
14 template<typename T>
15 void fPrintMatrix(vector<T> input){
16     for(auto x: input){
17         for(auto y: x){
18             cout << y << ",";
19         }
20         cout << endl;
21     }
22 }
23
24 template<typename T, typename T1>
25 void fPrintHashmap(unordered_map<T, T1> input){
26     for(auto x: input){
27         cout << format("{} , {}" | ", x.first, x.second);
28     }
29     cout << endl;
30 }
31
32 void fPrintBinaryTree(TreeNode* root){
33     // sending it back
34     if (root == nullptr) return;
35
36     // printing
37     PRINTLINE
38     cout << "root->val = " << root->val << endl;
39
40     // calling the children
41     fPrintBinaryTree(root->left);
42     fPrintBinaryTree(root->right);
43
44     // returning
45     return;
46 }
47
48
49 void fPrintLinkedList(ListNode* root){
50     if (root == nullptr) return;
51     cout << root->val << " -> ";
52     fPrintLinkedList(root->next);
53     return;
54 }
55
56 template<typename T>
57 void fPrintContainer(T input){
58     for(auto x: input) cout << x << ", ";
59     cout << endl;
60     return;
61 }
62 // =====
63 template <typename T>
64 auto size(std::vector<std::vector<T>> inputMatrix){
65     cout << format("{} , {} \n", inputMatrix.size(), inputMatrix[0].size());
66 }

```

```

67
68 template <typename T>
69 auto size(const std::string inputstring, std::vector<std::vector<T>> inputMatrix){
70     cout << format("{} = [ {}, {} ]\n", inputstring, inputMatrix.size(),
71     inputMatrix[0].size());
72 }

```

A.6 TimerClass.hpp

```

1 struct Timer
2 {
3     std::chrono::time_point<std::chrono::high_resolution_clock> startpoint;
4     std::chrono::time_point<std::chrono::high_resolution_clock> endpoint;
5     std::chrono::duration<long long, std::nano> duration;
6     std::string filename;
7     std::string functionname;
8
9     // constructor
10    Timer() {start();}
11    Timer(std::string logfile_arg): filename(std::move(logfile_arg)) {start();}
12    Timer(std::string logfile_arg,
13          std::string func_arg): filename(std::move(logfile_arg)),
14                                     functionname(std::move(func_arg)) {start();}
15
16    void start() {startpoint = std::chrono::high_resolution_clock::now();}
17    void stop() {endpoint = std::chrono::high_resolution_clock::now();
18    fetchtime();}
19
20    void fetchtime(){
21        duration = std::chrono::duration_cast<std::chrono::nanoseconds>(endpoint -
22        startpoint);
23        cout << format("{} nanoseconds \n", duration.count());
24    }
25    void fetchtime(string stringarg){
26        duration = std::chrono::duration_cast<std::chrono::nanoseconds>(endpoint -
27        startpoint);
28        cout << format("{} took {} nanoseconds \n", stringarg, duration.count());
29    }
30    void measure(){
31        auto temp = std::chrono::high_resolution_clock::now();
32        auto nsduration =
33        std::chrono::duration_cast<std::chrono::nanoseconds>(temp - startpoint);
34        auto msduration =
35        std::chrono::duration_cast<std::chrono::microseconds>(temp - startpoint);
36        auto sduration = std::chrono::duration_cast<std::chrono::seconds>(temp -
37        startpoint);
38        cout << format("{} nanoseconds | {} microseconds | {} seconds \n",
39        nsduration.count(), msduration.count(), sduration.count());
40    }
41    ~Timer(){
42        auto temp = std::chrono::high_resolution_clock::now();
43        auto nsduration =
44        std::chrono::duration_cast<std::chrono::nanoseconds>(temp - startpoint);

```

```

38     auto msduration =
std::chrono::duration_cast<std::chrono::microseconds>(temp - startpoint);
39     auto milliduration =
std::chrono::duration_cast<std::chrono::milliseconds>(temp - startpoint);
40     auto sduration = std::chrono::duration_cast<std::chrono::seconds>(temp -
startpoint);
41     PRINTLINE
42     cout << format("{} nanoseconds | {} microseconds | {} milliseconds | {}
seconds \n",
43         nsduration.count(), msduration.count(), milliduration.count(),
sduration.count());
44
45     // writing to the file
46     if (!filename.empty()){
47         std::ofstream fileobj(filename, std::ios::app);
48         if (fileobj){
49             if (functionname.empty()){
50                 fileobj << "main" << "," << nsduration.count() << "," <<
msduration.count() << "," << sduration.count() << "\n";
51             }
52             else{
53                 fileobj << functionname << "," << nsduration.count() << "," <<
msduration.count() << "," << sduration.count() << "\n";
54             }
55         }
56     }
57 }
58 };

```

A.7 utils.hpp

```

1  // =====
2  #include "svr_WriteToCSV.hpp"
3  // =====
4  template <typename F, typename R>
5  constexpr auto fElementWise(F&& func, R& range){
6      std::transform(std::begin(range),
7                    std::end(range),
8                    std::begin(range),
9                    std::forward<F>(func));
10     // return range;
11 }
12 // =====
13 #include "svr_repmat.hpp"
14 // =====
15 auto SineElementWise(auto& input, auto constantvalue){
16     for(auto& x: input) {x = std::sin(constantvalue * x);}
17     // replace this with std::transform
18 };
19 // =====
20 #include "svr_linspace.hpp"
21 // =====
22 #include "svr_fft.hpp"

```

```

23 // =====
24 template <typename T>
25 auto abs(vector<complex<T>> inputvector){
26     vector<T> temp(inputvector.size(), 0);
27     std::transform(temp.begin(),
28                   temp.end(),
29                   temp.begin(),
30                   [](T a){return std::abs(a);});
31     return temp;
32 }
33 // =====
34 #include "svr_rand.hpp"
35 // =====
36 #include "svr_operator_star.hpp"
37 // =====
38 #include "svr_operators.hpp"
39 // =====
40 #include "svr_tensor_inits.hpp"
41 // =====
42 #include "svr_sin.hpp"
43 // =====
44 #include "svr_slice.hpp"
45 // =====
46 #include "svr_matrix_operations.hpp"
47 // =====
48 #include <boost/type_index.hpp>
49 template <typename T>
50 auto type(T inputarg){
51     std::cout <<
52     boost::typeindex::type_id_with_cvr<decltype(inputarg)>().pretty_name()<< "\n";
53 }
54 // =====
55 #include "svr_shape.hpp"
56 // =====
57 #include "svr_sum.hpp"

```

A.8 svr_WriteToCSV.hpp

```

1 // =====
2 template <typename T>
3 void fWriteVector(const vector<T>&          inputvector,
4                  const string&             filename){
5
6     // opening a file
7     std::ofstream fileobj(filename);
8     if (!fileobj) {return;}
9
10    // writing the real parts in the first column and the imaginary parts in the
11    // second column
12    if constexpr(std::is_same_v<T, std::complex<double>> ||
13                  std::is_same_v<T, std::complex<float>> ||
14                  std::is_same_v<T, std::complex<long double>>){
15        for(int i = 0; i<inputvector.size(); ++i){

```



```

15         // adding entry
16         fileobj << inputvector[i].real() << "+" << inputvector[i].imag() << "i";
17
18         // adding delimiter
19         if(i!=inputvector.size()-1) {fileobj << ",";}
20         else {fileobj << "\n";}
21     }
22 }
23 else{
24     for(int i = 0; i<inputvector.size(); ++i){
25         fileobj << inputvector[i];
26         if(i!=inputvector.size()-1) {fileobj << ",";}
27         else {fileobj << "\n";}
28     }
29 }
30
31 // return
32 return;
33 }
34 // Matrix writing =====
35 template <typename T>
36 auto fWriteMatrix(const std::vector<std::vector<T>> inputMatrix,
37                  const string filename){
38
39     // opening a file
40     std::ofstream fileobj(filename);
41
42     // writing
43     if (fileobj){
44         for(int i = 0; i<inputMatrix.size(); ++i){
45             for(int j = 0; j<inputMatrix[0].size(); ++j){
46                 fileobj << inputMatrix[i][j];
47                 if (j!=inputMatrix[0].size()-1) {fileobj << ",";}
48                 else {fileobj << "\n";}
49             }
50         }
51     }
52     else{
53         cout << format("File-write to {} failed\n", filename);
54     }
55 }
56
57
58 template <>
59 auto fWriteMatrix(const std::vector<std::vector<std::complex<double>>> inputMatrix,
60                  const string filename){
61
62     // opening a file
63     std::ofstream fileobj(filename);
64
65     // writing
66     if (fileobj){
67         for(int i = 0; i<inputMatrix.size(); ++i){
68             for(int j = 0; j<inputMatrix[0].size(); ++j){
69                 fileobj << inputMatrix[i][j].real() << "+" <<
inputMatrix[i][j].imag() << "i";
70                 if (j!=inputMatrix[0].size()-1) {fileobj << ",";}
71                 else {fileobj << "\n";}

```

```

72     }
73 }
74 }
75 else{
76     cout << format("File-write to {} failed\n", filename);
77 }
78 }

```

A.9 svr_repmat.hpp

```

1  template<typename T>
2  constexpr auto repmat(const vector<vector<T>>& input,
3                       const vector<int> dimensions){
4
5      // calculating resulting dimensions
6      auto numrows {static_cast<int>(input.size()) * dimensions[0]};
7      auto numcols {static_cast<int>(input[0].size()) * dimensions[1]};
8
9      // creating new matrix
10     vector<vector<T>> finaloutput;
11     vector<T> temp;
12     auto sourcerow {-1};
13     auto sourcecol {-1};
14     for(int i = 0; i<numrows; ++i){
15         temp.clear();
16         for(int j = 0; j<numcols; ++j){
17             sourcerow = i % static_cast<int>(input.size());
18             sourcecol = j % static_cast<int>(input[0].size());
19             temp.push_back(input[sourcerow][sourcecol]);
20         }
21         finaloutput.push_back(temp);
22     }
23
24     // returning the final output
25     return finaloutput;
26
27 };
28
29 template <typename T>
30 constexpr auto repmat(const vector<T>& input,
31                      const vector<int> dimensions){
32
33     // calculating resulting dimensions
34     auto numrows {static_cast<int>(dimensions[0])};
35     auto numcols {static_cast<int>(input.size()) * dimensions[1]};
36
37     // creating new matrix
38     vector<vector<T>> finaloutput;
39     vector<T> temp;
40
41     // filling up the vector
42     auto sourcerow {-1};
43     auto sourcecol {-1};

```

```

44     for(int i = 0; i<numrows; ++i){
45         temp.clear();
46         for(int j = 0; j<numcols; ++j){
47             sourcerow = i % 1;
48             sourcecol = j % static_cast<int>(input.size());
49             temp.push_back(input[sourcecol]);
50         }
51         finaloutput.push_back(temp);
52     }
53
54     // returning the final output
55     return finaloutput;
56
57 };

```

A.10 svr_linspace.hpp

```

1  // in-place
2  template <typename T>
3  auto linspace(auto&          input,
4                auto          startvalue,
5                auto          endvalue,
6                auto          numpoints) -> void
7  {
8      auto stepsize = static_cast<T>(endvalue -
9      startvalue)/static_cast<T>(numpoints-1);
10     for(int i = 0; i<input.size(); ++i) {input[i] = startvalue + i*stepsize;}
11 };
12 // in-place
13 template <typename T>
14 auto linspace(vector<complex<T>>& input,
15               auto          startvalue,
16               auto          endvalue,
17               auto          numpoints) -> void
18 {
19     auto stepsize = static_cast<T>(endvalue -
20     startvalue)/static_cast<T>(numpoints-1);
21     for(int i = 0; i<input.size(); ++i) {
22         input[i] = startvalue + static_cast<T>(i)*stepsize;
23     }
24 };
25 // return-type
26 template <typename T>
27 auto linspace(T          startvalue,
28               T          endvalue,
29               size_t      numpoints)
30 {
31     vector<T> input(numpoints);
32     auto stepsize = static_cast<T>(endvalue -
33     startvalue)/static_cast<T>(numpoints-1);

```

```

33     for(int i = 0; i<input.size(); ++i) {input[i] = startvalue +
        static_cast<T>(i)*stepsize;}
34
35     return input;
36 };
37
38 // return-type
39 template <typename T, typename U>
40 auto linspace(T          startvalue,
41              U          endvalue,
42              size_t      numpoints)
43 {
44     vector<double> input(numpoints);
45     auto stepsize = static_cast<double>(endvalue -
        startvalue)/static_cast<double>(numpoints-1);
46
47     for(int i = 0; i<input.size(); ++i) {input[i] = startvalue + i*stepsize;}
48
49     return input;
50 };

```

A.11 svr_fft.hpp

```

1 // =====
2 template<typename T>
3 auto fft(vector<T> inputarray){
4
5     // building just the time thing
6     vector<complex<T>> basiswithoutfrequency(inputarray.size(), 0);
7     linspace(basiswithoutfrequency,
8             0,
9             basiswithoutfrequency.size()-1,
10            basiswithoutfrequency.size());
11
12     auto lambda = \
13         [&basiswithoutfrequency](const complex<T> arg){
14         return std::exp(-1.00 * \
15             std::complex<T>{0, 1} * \
16             2.00 * std::numbers::pi * \
17             static_cast<complex<T>>(arg) \
18             / static_cast<complex<T>>(basiswithoutfrequency.size()));
19     };
20     std::transform(basiswithoutfrequency.begin(),
21                   basiswithoutfrequency.end(),
22                   basiswithoutfrequency.begin(),
23                   lambda);
24
25     // building basis vectors
26     vector<vector<complex<T>>> basisvectors;
27     for(auto i = 0; i < inputarray.size(); ++i){
28
29         // making a copy of the basis-without-frequency
30         auto temp = basiswithoutfrequency;

```

```

31
32     // exponentiating with associated frequency
33     std::transform(temp.begin(),
34                   temp.end(),
35                   temp.begin(),
36                   [i](const auto arg1){
37                       return std::pow(arg1, i);
38                   });
39
40     // pushing to end of basis vectors
41     basisvectors.push_back(std::move(temp));
42 }
43
44 // building coefficient arrays
45 vector<complex<double>> finaloutput(inputarray.size(), 0);
46 finaloutput.reserve(finaloutput.size());
47
48 // producing inner-products
49 for(int i = 0; i<inputarray.size(); ++i){
50     finaloutput[i] = \
51         std::inner_product(basisvectors[i].begin(),
52                           basisvectors[i].end(),
53                           inputarray.begin(),
54                           complex<double>(0),
55                           std::plus<std::complex<double>>(),
56                           [&inputarray](complex<double> a, T b){
57                               return a * static_cast<complex<double>>(b) /
58                               static_cast<double>(std::sqrt(inputarray.size()));
59                           });
60 }
61
62 // returning finaloutput
63 return finaloutput;
64 }
65 // =====
66 template<typename T>
67 auto fft(vector<T> inputarray, size_t nfft){
68
69     // throwing an error
70     if (nfft < inputarray.size()) {std::cerr << "size-mismatch\n";}
71
72     // building time-only basis
73     vector<complex<T>>
74     basiswithoutfrequency = linspace(static_cast<std::complex<T>>(0),
75                                     static_cast<std::complex<T>>(nfft-1),
76                                     nfft);
77
78     auto lambda = [&basiswithoutfrequency](const complex<T> arg){
79         return std::exp(-1.00 * complex<double>{0, 1} * 2.00 * \
80             std::numbers::pi * static_cast<complex<T>>(arg) / \
81             static_cast<complex<T>>(basiswithoutfrequency.size()));
82     };
83
84     std::transform(basiswithoutfrequency.begin(),
85                   basiswithoutfrequency.end(),
86                   basiswithoutfrequency.begin(),
87                   lambda);
88
89     // building basis vectors

```

```

88     vector<vector<complex<double>>> basisvectors;
89     for(auto i = 0; i < inputarray.size(); ++i){
90
91         // making a copy of the basis-without-frequency
92         vector<complex<double>> temp = basiswithoutfrequency;
93
94         // exponentiating with associated frequency
95         std::transform(temp.begin(),
96                        temp.end(),
97                        temp.begin(),
98                        [i](const auto arg1){return std::pow(arg1, i);});
99
100        // pushing to end of basis vectors
101        basisvectors.push_back(std::move(temp));
102    }
103
104
105    // building the projection
106    vector<complex<double>> finaloutput(inputarray.size(), 0);
107    finaloutput.reserve(finaloutput.size());
108    #pragma omp parallel for
109    for(int i = 0; i<inputarray.size(); ++i){
110        // writing coefficients
111        finaloutput[i] = \
112            std::inner_product(basisvectors[i].begin(),
113                               basisvectors[i].end(),
114                               inputarray.begin(),
115                               complex<double>(0),
116                               std::plus<std::complex<double>>(),
117                               [&nfft](const complex<double> a,
118                                       const T b){
119                                   return a * static_cast<complex<double>>(b) /
120                                   static_cast<double>(std::sqrt(nfft));
121                               });
122    }
123
124    // returning finaloutput
125    return finaloutput;
126 }
127 // =====
128 template <>
129 auto fft(vector<complex<double>> inputarray){
130
131     // aliasing
132     using T = double;
133
134     // building time-only basis-vector
135     vector<complex<T>>
136     basiswithoutfrequency = linspace(std::complex<T>(0),
137                                     std::complex<T>(inputarray.size()-1),
138                                     inputarray.size());
139     auto lambda = [&basiswithoutfrequency](const complex<T> arg){
140         return std::exp(-1.00 * complex<double>{0, 1} * 2.00 * \
141                     std::numbers::pi * static_cast<complex<T>>(arg) \
142                     / static_cast<complex<T>>(basiswithoutfrequency.size()));
143     };
144     std::transform(basiswithoutfrequency.begin(),

```

```

145         basiswithoutfrequency.end(),
146         basiswithoutfrequency.begin(),
147         lambda);
148
149
150     // building basis vectors
151     vector<vector<complex<T>>> basisvectors;
152     for(auto i = 0; i < inputarray.size(); ++i){
153
154         // making a copy of the basis-without-frequency
155         vector<complex<T>> temp = basiswithoutfrequency;
156
157         // adding frequency component
158         std::transform(temp.begin(),
159                        temp.end(),
160                        temp.begin(),
161                        [i](const auto arg1){return std::pow(arg1, i);});
162
163         // pushing to end of basis vectors
164         basisvectors.push_back(std::move(temp));
165     }
166
167
168     // building the coefficients
169     vector< complex<T> > finaloutput(inputarray.size(), 0);
170     finaloutput.reserve(finaloutput.size());
171     for(int i = 0; i<inputarray.size(); ++i)
172         finaloutput[i] = std::inner_product(basisvectors[i].begin(),
173                                            basisvectors[i].end(),
174                                            inputarray.begin(),
175                                            std::complex<double>(0));
176
177     // scaling down the coefficients
178     std::transform(finaloutput.begin(),
179                   finaloutput.end(),
180                   finaloutput.begin(),
181                   [&inputarray](auto argx){
182                       return argx / static_cast<T>(std::sqrt(inputarray.size()));
183                   });
184
185
186     // returning finaloutput
187     return finaloutput;
188 }
189
190 // =====
191 template<typename T>
192 auto fft(std::vector<std::vector<T>> inputMatrix,
193         int nfft){
194
195     // initializing
196     std::vector<std::vector<std::complex<T>>>
197     finaloutput(inputMatrix.size(),
198                 std::vector<std::complex<T>>(inputMatrix[0].size(), 0));
199
200     // checking if we need to pad the rows
201     if (inputMatrix[0].size() > nfft) {std::cerr << "nfft < row-size\n";}
202     else if (inputMatrix[0].size() < nfft) {

```

```

203
204     // creating a placeholder
205     std::vector<std::vector<std::complex<T>>>
206     temp(inputMatrix.size(),
207          std::vector<std::complex<T>>(nfft, 0));
208
209     // moving to the finaloutput
210     finaloutput.clear();
211     finaloutput = std::move(temp);
212
213 }
214
215 // filling final-output with the input-values
216 for(int i = 0; i<inputMatrix.size(); ++i)
217     std::copy(inputMatrix[i].begin(),
218              inputMatrix[i].end(),
219              finaloutput[i].begin());
220
221 // performing fft
222 #pragma omp parallel for
223 for(auto& row: finaloutput)    {row = fft(row);}
224
225 // returning the matrix
226 return finaloutput;
227
228 }
229 // =====
230 template<>
231 auto fft(std::vector<std::vector<std::complex<double>>> inputMatrix,
232         int nfft){
233
234     // changing types
235     using T = double;
236
237     // initializing
238     std::vector<std::vector<std::complex<T>>>
239     finaloutput(inputMatrix.size(),
240                std::vector<std::complex<T>>(inputMatrix[0].size(), complex<T>(0)));
241
242     // checking if we need to pad the rows
243     if (inputMatrix[0].size() > nfft) {std::cerr << "nfft < row-size\n";}
244     else if (inputMatrix[0].size() < nfft)    {
245
246         // creating a placeholder
247         std::vector<std::vector<std::complex<T>>>
248         temp(inputMatrix.size(),
249              std::vector<std::complex<T>>(nfft, std::complex<T>(0)));
250
251         // moving to the finaloutput
252         finaloutput.clear();
253         finaloutput = std::move(temp);
254     }
255
256     // filling final-output with the input-values
257     for(int i = 0; i<inputMatrix.size(); ++i){
258         std::copy(inputMatrix[i].begin(),
259                  inputMatrix[i].end(),
260                  finaloutput[i].begin());

```



```

261     }
262
263     // performing fft
264     for(int i = 0; i<finaloutput.size(); ++i)
265         finaloutput[i] = std::move(fft(finaloutput[i]));
266
267     // returning the matrix
268     return finaloutput;
269
270 }
271 // ifft (vector) =====
272 template <typename T>
273 auto ifft(const std::vector<T> inputvector){
274
275     // setting up data type
276     using T2 = std::conditional_t<std::is_same_v<T, std::complex<float>>,
277                                     std::complex<float>,
278                                     std::complex<double>>;
279     using T3 = typename T2::value_type;
280
281     // building basis
282     vector<T2>
283     basiswithoutfrequency {linspace(static_cast<T2>(0),
284                                     static_cast<T2>(inputvector.size()-1),
285                                     inputvector.size())};
286
287     // lambda for building basis without frequency
288     auto lambda = \
289         [&basiswithoutfrequency](const T2 arg){
290             return std::exp(1.00 * T2{0, 1} * 2.00 * \
291                             std::numbers::pi * arg / \
292                             static_cast<T2>(basiswithoutfrequency.size()));
293     };
294
295     // building the basis without frequency
296     std::transform(basiswithoutfrequency.begin(),
297                   basiswithoutfrequency.end(),
298                   basiswithoutfrequency.begin(),
299                   lambda);
300
301     // building basis vectors
302     std::vector<std::vector<T2>> bases;
303     for(int i = 0; i < inputvector.size(); ++i){
304
305         // creating bases with frequency components
306         auto basiswithfrequency = basiswithoutfrequency;
307         std::transform(basiswithfrequency.begin(),
308                       basiswithfrequency.end(),
309                       basiswithfrequency.begin(),
310                       [i](T2 argx){
311                             return static_cast<T2>(std::pow(argx,
312 static_cast<T3>(i)));
313                         });
314
315         // pushing to the basis vectors
316         bases.push_back(std::move(basiswithfrequency));
317     }

```

```

318 // computing projections
319 std::vector<T2> projection_coefficients(inputvector.size());
320 for(auto i = 0; i < bases.size(); ++i){
321
322     // calculating inner-product
323     auto temp {std::inner_product(bases[i].begin(), bases[i].end(),
324                                   inputvector.begin(),
325                                   static_cast<T2>(0),
326                                   std::plus<T2>(),
327                                   [&inputvector](auto arg_bases, auto
arg_inputvector){
328                                     return static_cast<T2>(arg_bases) *
\
329                                     static_cast<T2>(arg_inputvector)
/ \
330 static_cast<T3>(std::sqrt(inputvector.size())));
331                                     }));
332
333     // // calculating inner-product
334     // auto temp {std::inner_product(bases[i].begin(), bases[i].end(),
335     //                                inputvector.begin(),
336     //                                static_cast<T2>(0),
337     //                                std::plus<T2>(),
338     //                                [&inputvector](auto arg_bases, auto
arg_inputvector){
339     //                                    return static_cast<T2>(arg_bases) *
\
340     //                                    static_cast<T2>(arg_inputvector) / \
341     //                                    static_cast<T3>((inputvector.size()));
342     //                                    }));
343
344     // writing to the final output
345     projection_coefficients[i] = std::move(temp);
346 }
347
348 // returning the coefficients
349 return projection_coefficients;
350
351 }

```

A.12 svr_rand.hpp

```

1 // =====
2 template <typename T>
3 auto rand(const T min, const T max) {
4     static std::random_device rd; // Seed
5     static std::mt19937 gen(rd()); // Mersenne Twister generator
6     std::uniform_real_distribution<> dist(min, max);
7     return dist(gen);
8 }

```

```

9 // =====
10 template <typename T>
11 auto rand(const T      min,
12           const T      max,
13           const size_t  numelements) {
14     static std::random_device rd; // Seed
15     static std::mt19937 gen(rd()); // Mersenne Twister generator
16     std::uniform_real_distribution<> dist(min, max);
17
18     // building the finaloutput
19     vector<T> finaloutput(numelements);
20     for(int i = 0; i<finaloutput.size(); ++i) {finaloutput[i] =
21         static_cast<T>(dist(gen));}
22
23     return finaloutput;
24 }
25 // =====
26 template <typename T>
27 auto rand(const T      argmin,
28           const T      argmax,
29           const vector<int> dimensions){
30
31     // throwing an error if dimension is greater than two
32     if (dimensions.size() > 2) {std::cerr << "dimensions are too high\n";}
33
34     // creating random engine
35     static std::random_device rd; // Seed
36     static std::mt19937 gen(rd()); // Mersenne Twister generator
37     std::uniform_real_distribution<> dist(argmin, argmax);
38
39     // building the finaloutput
40     vector<vector<T>> finaloutput;
41     for(int i = 0; i<dimensions[0]; ++i){
42         vector<T> temp;
43         for(int j = 0; j<dimensions[1]; ++j) {temp.push_back(dist(gen));}
44         // cout << format("\t\t temp = {}\n", temp);
45
46         finaloutput.push_back(temp);
47     }
48
49     // returning the finaloutput
50     return finaloutput;
51 }
52 // =====
53 auto rand(const vector<int> dimensions){
54
55     using ReturnType = double;
56
57     // throwing an error if dimension is greater than two
58     if (dimensions.size() > 2) {std::cerr << "dimensions are too high\n";}
59
60     // creating random engine
61     static std::random_device rd; // Seed
62     static std::mt19937 gen(rd()); // Mersenne Twister generator
63     std::uniform_real_distribution<> dist(0.00, 1.00);
64
65     // building the finaloutput

```

```

66     vector<vector<ReturnType>> finaloutput;
67     for(int i = 0; i<dimensions[0]; ++i){
68         vector<ReturnType> temp;
69         for(int j = 0; j<dimensions[1]; ++j)    {temp.push_back(dist(gen));}
70         finaloutput.push_back(std::move(temp));
71     }
72
73     // returning the finaloutput
74     return finaloutput;
75
76 }
77 // =====
78 template <typename T>
79 auto rand_complex_double(const T          argmin,
80                          const T          argmax,
81                          const vector<int>& dimensions){
82
83     // throwing an error if dimension is greater than two
84     if (dimensions.size() > 2) {std::cerr << "dimensions are too high\n";}
85
86     // creating random engine
87     static std::random_device rd;    // Seed
88     static std::mt19937 gen(rd());   // Mersenne Twister generator
89     std::uniform_real_distribution<> dist(argmin, argmax);
90
91     // building the finaloutput
92     vector<vector<complex<double>>> finaloutput;
93     for(int i = 0; i<dimensions[0]; ++i){
94         vector<complex<double>> temp;
95         for(int j = 0; j<dimensions[1]; ++j)
96         {temp.push_back(static_cast<double>(dist(gen)));}
97         finaloutput.push_back(std::move(temp));
98     }
99
100    // returning the finaloutput
101    return finaloutput;
102 }

```

A.13 svr_operator_star.hpp

```

1  // scalar * vector =====
2  template <typename T>
3  auto operator*(T          scalar,
4                 const vector<T>& inputvector){
5      vector<T> temp(inputvector.size());
6      std::transform(inputvector.begin(),
7                     inputvector.end(),
8                     temp.begin(),
9                     [&scalar](T x){return scalar * x;});
10     return temp;
11 }
12

```

```

13 template <typename T1, typename T2>
14 auto operator*(T1 scalar,
15               const vector<T2>& inputvector){
16     using T3 = decltype(std::declval<T1>() * std::declval<T2>());
17     vector<T3> temp(inputvector.size());
18     std::transform(inputvector.begin(),
19                   inputvector.end(),
20                   temp.begin(),
21                   [&scalar](auto x){return static_cast<T3>(scalar) *
22 static_cast<T3>(x);});
23     return temp;
24 }
25 // // template <>
26 // auto operator*(double doublescalar,
27 //               std::vector<std::complex<double>> argvector){
28 //
29 //     std::vector<std::complex<double>> temp(argvector.size());
30 //     std::transform(argvector.begin(),
31 //                   argvector.end(),
32 //                   temp.begin(),
33 //                   [&doublescalar](complex<double> x){return
34 // static_cast<complex<double>>(doublescalar) * x;});
35 //     return temp;
36 // }
37 // vector * scalar =====
38 template <typename T>
39 auto operator*(const vector<T>& inputvector,
40               T scalar){
41     vector<T> temp(inputvector.size());
42     std::transform(inputvector.begin(), inputvector.end(), temp.begin(),
43                   [&scalar](T x){return scalar * x;});
44     return temp;
45 }
46 // scalar * matrix =====
47 template <typename T>
48 auto operator*(T scalar,
49               const std::vector<std::vector<T>>& inputMatrix){
50     std::vector<std::vector<T>> temp {inputMatrix};
51     for(int i = 0; i<inputMatrix.size(); ++i){
52         std::transform(inputMatrix[i].begin(),
53                       inputMatrix[i].end(),
54                       temp[i].begin(),
55                       [&scalar](T x){return scalar * x;});
56     }
57     return temp;
58 }
59 // scalar * matrix =====
60 template <typename T1, typename T2>
61 auto operator*(T1 scalar,
62               const std::vector<std::vector<T2>>& inputMatrix){
63     std::vector<std::vector<T2>> temp {inputMatrix};
64     for(int i = 0; i<inputMatrix.size(); ++i){
65         std::transform(inputMatrix[i].begin(),
66                       inputMatrix[i].end(),
67                       temp[i].begin(),

```

```

68         [&scalar](T2 x){return static_cast<T2>(scalar) * x;});
69     }
70     return temp;
71 }
72
73 // matrix * matrix =====
74 template <typename T>
75 auto operator*(const std::vector<std::vector<T>>& matA,
76               const std::vector<std::vector<T>>& matB)
77 {
78
79     // throwing error
80     if (matA.size() != matB.size() || matA[0].size() != matB[0].size())
81     {std::cerr << "size issues\n";}
82
83     // creating placeholder
84     auto temp {matA};
85
86     // performing multiplication
87     for(int i = 0; i<matA.size(); ++i){
88         for(int j = 0; j<matA[0].size(); ++j){
89             temp[i][j] *= matB[i][j];
90         }
91     }
92
93     // returning
94     return temp;
95 }
96 // matrix-multiplication =====
97 template <typename T1, typename T2>
98 auto matmul(const std::vector<std::vector<T1>>& matA,
99            const std::vector<std::vector<T2>>& matB)
100 {
101
102     // throwing error
103     if (matA[0].size() != matB.size()) {std::cerr << "dimension-mismatch \n";}
104
105     // getting result-type
106     using ResultType = decltype(std::declval<T1>() * std::declval<T2>() + \
107                                std::declval<T1>() * std::declval<T2>() );
108
109     // creating aliases
110     auto finalnumrows {matA.size()};
111     auto finalnumcols {matB[0].size()};
112
113     // creating placeholder
114     auto rowcolproduct = [&](auto rowA, auto colB){
115         ResultType temp {0};
116         for(int i = 0; i < matA.size(); ++i) {temp +=
117             static_cast<ResultType>(matA[rowA][i]) +
118             static_cast<ResultType>(matB[i][colB]);}
119         return temp;
120     };
121
122     // producing row-column combinations
123     std::vector<std::vector<ResultType>> finaloutput(finalnumrows,
124     std::vector<ResultType>(finalnumcols));

```

```

121     for(int row = 0; row < finalnumrows; ++row){for(int col = 0; col <
122         finalnumcols; ++col){finaloutput[row][col] = rowcolproduct(row, col);}}
123
124     // returning
125     return finaloutput;
126 }
127 // =====

```

A.14 svr_operators.hpp

```

1  template <typename T>
2  std::vector<T> operator+(const std::vector<T>& a, const std::vector<T>& b) {
3      // Identify which is bigger
4      const auto& big = (a.size() > b.size()) ? a : b;
5      const auto& small = (a.size() > b.size()) ? b : a;
6
7      std::vector<T> result = big; // copy the bigger one
8
9      // Add elements from the smaller one
10     for (size_t i = 0; i < small.size(); ++i) {
11         result[i] += small[i];
12     }
13
14     return result;
15 }
16 template <typename T>
17 std::vector<T>& operator+=(std::vector<T>& a, const std::vector<T>& b) {
18
19     const auto& small = (a.size() < b.size()) ? a : b;
20     const auto& big = (a.size() < b.size()) ? b : a;
21
22     // If b is bigger, resize 'a' to match
23     if (a.size() < b.size()) {a.resize(b.size());}
24
25     // Add elements
26     for (size_t i = 0; i < small.size(); ++i) {a[i] += b[i];}
27
28     // returning elements
29     return a;
30 }
31 template <typename T>
32 std::vector<std::vector<T>> operator+(const std::vector<std::vector<T>>& a,
33                                     const std::vector<std::vector<T>>& b)
34 {
35     // throwing an error if dimension error occurs
36     if ((a.size() != b.size()) || (a[0].size() != b[0].size())) {
37         cout << format("a.dimensions = [{},{}], b.shape = [{},{}]\n",
38             a.size(), a[0].size(), b.size(), b[0].size());
39         std::cerr << "dimensions don't match\n";
40     }
41
42     // performing the addition

```

```

44     auto temp    {a};
45     for(int i = 0; i<b.size(); ++i){
46         for(int j = 0; j<b[0].size(); ++j){
47             temp[i][j] += b[i][j];
48         }
49     }
50
51     // returning
52     return temp;
53 }
54 // =====
55 // Aim: subtracting scalar from a vector
56 template <typename T>
57 std::vector<T> operator-(const std::vector<T>& a, const T scalar){
58     std::vector<T> temp(a.size());
59     std::transform(a.begin(),
60                   a.end(),
61                   temp.begin(),
62                   [scalar](T x){return (x - scalar);});
63     return temp;
64 }
65 // scalar operators =====
66 auto operator*(const std::complex<double>    complexscalar,
67                const double                  doublescalar){
68     return complexscalar * static_cast<std::complex<double>>(doublescalar);
69 }
70 auto operator*(const double                  doublescalar,
71                const std::complex<double>    complexscalar){
72     return complexscalar * static_cast<std::complex<double>>(doublescalar);
73 }
74 auto operator*(const std::complex<double>    complexscalar,
75                const int                      scalar){
76     return complexscalar * static_cast<std::complex<double>>(scalar);
77 }
78 auto operator*(const int                      scalar,
79                const std::complex<double>    complexscalar){
80     return complexscalar * static_cast<std::complex<double>>(scalar);
81 }

```

A.15 svr_tensor_inits.hpp

```

1  std::vector<std::vector<double>> Zeros(vector<int> dimensions){
2
3      // throwing an error if the dimension is more than 2
4      if (dimensions.size() > 2) {std::cerr << "Dimensions are a little too much";}
5
6      // building the vector
7      std::vector<std::vector<double>> temp;
8      for(int i = 0; i<dimensions[0]; ++i){
9          temp.emplace_back(vector<double>(dimensions[1], 0));
10     };
11
12     // returning the output

```



```

13     return temp;
14 }
15
16 auto Zeros_complex_double(vector<int> dimensions){
17
18     // throwing an error if the dimension is more than 2
19     if (dimensions.size() > 2) {std::cerr << "Dimensions are a little too much";}
20
21     // building the vector
22     std::vector<std::vector<std::complex<double>>> temp;
23     for(int i = 0; i<dimensions[0]; ++i){
24         temp.emplace_back(std::vector<std::complex<double>>(dimensions[1],
25             std::complex<double>{0,0}));
26     };
27
28     // returning the output
29     return temp;
30 }
31
32 // =====
33 std::vector<std::vector<double>> Ones(vector<int> dimensions){
34
35     // throwing an error if the dimension is more than 2
36     if (dimensions.size() > 2) {std::cerr << "Dimensions are a little too much";}
37
38     // building the vector
39     std::vector<std::vector<double>> temp;
40     for(int i = 0; i<dimensions[0]; ++i){
41         temp.emplace_back(vector<double>(dimensions[1], 1));
42     };
43
44     // returning the output
45     return temp;
46 }

```

A.16 svr_sin.hpp

```

1  template <typename T>
2  auto sin(vector<T> input){
3      auto temp {input};
4      std::transform(input.begin(),
5                     input.end(),
6                     temp.begin(),
7                     [](const T x){return std::sin(x);});
8      return temp;
9  }
10
11 template <typename T>
12 auto sin_inplace(vector<T>& input) -> void
13 {
14     std::transform(input.begin(),
15                    input.end(),
16                    input.begin(),

```

```

17         [](const T x){return std::sin(x);});
18     }
19
20     // =====
21     template <typename T>
22     auto cosd(T input){
23         return std::cos(input * std::numbers::pi / 180);
24     }

```

A.17 svr_slice.hpp

```

1  template<typename T>
2  auto slice(const std::vector<std::vector<T>>&    inputMatrix,
3             vector<int>                          arglist)
4  {
5
6      // updating rows and columns
7      if (arglist[0] == -1)    {arglist[0] = 0;}
8      else if (arglist[0] == -2) {arglist[0] = inputMatrix.size()-1;}
9      if (arglist[1] == -1)    {arglist[1] = 0;}
10     else if (arglist[1] == -2) {arglist[1] = inputMatrix.size()-1;}
11     if (arglist[2] == -1)    {arglist[2] = 0;}
12     else if (arglist[2] == -2) {arglist[2] = inputMatrix[0].size()-1;}
13     if (arglist[3] == -1)    {arglist[3] = 0;}
14     else if (arglist[3] == -2) {arglist[3] = inputMatrix[0].size()-1;}
15
16     // storing dimension values
17     int rowsize    {arglist[1] - arglist[0] + 1};
18     int colsize    {arglist[3] - arglist[2] + 1};
19
20     // building the final-output matrix
21     std::vector<std::vector<T>> finaloutput;
22     for(int row = arglist[0]; row <= arglist[1]; ++row){
23
24         // creating empty sub-row
25         vector<T> temp(colsize, 0);
26
27         // copying corresponding region to subrow
28         std::copy(inputMatrix[row].begin() + arglist[2],
29                 inputMatrix[row].begin() + arglist[3]+1,
30                 temp.begin());
31
32         // pushing to final output
33         finaloutput.push_back(std::move(temp));
34     }
35
36     // returning the final output
37     return finaloutput;
38 }

```

A.18 svr_matrix_operations.hpp

```

1 // =====
2 template <typename T>
3 auto dotproduct(const vector<T> argx,
4                const vector<T> argy)
5 {
6     // dimension checks
7     if (argx.size() != argy.size()) {std::cerr << "size disparity\n";}
8
9     // accumulating
10    T temp = 0;
11    for(int i = 0; i<argx.size(); ++i){
12
13        if constexpr(std::is_same_v<T, std::complex<double>> ||
14                    std::is_same_v<T, std::complex<float>> ||
15                    std::is_same_v<T, std::complex<long double>>){
16            temp += std::conj(argx[i]) * argy[i];
17        }
18        else{
19            temp += argx[i] * argy[i];
20        }
21    }
22
23
24    return temp;
25 }

```

A.19 svr_shape.hpp

```

1 // returning shape as vector =====
2 template <typename T>
3 auto shape(const T& inputTensor){
4
5     using U = std::decay_t<T>;
6
7     // type-traits checking
8     if constexpr (std::is_same_v<U, std::vector<double>> ||
9                 std::is_same_v<U, std::vector<float>> ||
10                std::is_same_v<U, std::vector<int>> ||
11                std::is_same_v<U, std::vector<complex<double>>> ||
12                std::is_same_v<U, std::vector<complex<float>>> ||
13                std::is_same_v<U, std::vector<complex<int>>> ||
14                std::is_same_v<U, std::string>){
15        return std::vector<int>{static_cast<int>(inputTensor.size())};
16    }
17    else if constexpr (std::is_same_v<U, std::vector<std::vector<double>>> ||
18                    std::is_same_v<U, std::vector<std::vector<float>>> ||
19                    std::is_same_v<U, std::vector<std::vector<int>>> ||
20                    std::is_same_v<U,
21                std::vector<std::vector<std::complex<double>>>> ||
22                    std::is_same_v<U,
23                std::vector<std::vector<std::complex<float>>>> ||

```

```

22         std::is_same_v<U,
std::vector<std::vector<std::complex<int>>>>)>){
23         return std::vector<int>{static_cast<int>(inputTensor.size()),
24                                static_cast<int>(inputTensor[0].size())};
25     }
26 }

```

A.20 svr_sum.hpp

```

1  template <size_t Axis, typename T>
2  auto sum(const vector<vector<T>>& inputMatrix){
3
4      // asserting dimensions for now
5      static_assert(Axis == 0 || Axis == 1, "Axis must be 0 or 1\n");
6
7      // splitting based on dimensions
8      if constexpr (Axis == 0){
9          auto returnTensor {std::vector<T>(inputMatrix[0].size(), 0.00)};
10         for(auto row = 0; row < inputMatrix.size(); ++row){
11             std::transform(inputMatrix[row].begin(),
12                            inputMatrix[row].end(),
13                            returnTensor.begin(),
14                            returnTensor.begin(),
15                            std::plus<T>{});
16         }
17         return returnTensor;
18     }
19     else{
20         auto returnTensor {std::vector<std::vector<T>>(inputMatrix.size(),
21                                                         std::vector<T>(1, 0.00))};
22         for(auto row = 0; row < inputMatrix.size(); ++row){
23             auto temp {std::accumulate(inputMatrix[row].begin(),
24                                        inputMatrix[row].end(),
25                                        T{0})};
26             returnTensor[row][0] = temp;
27         }
28         return returnTensor;
29     }
30 }

```

Appendix B

Octave Function Definitions

B.1 fReadCSV.m

```

1  function finalmatrix = fReadCSV(filename_string)
2      entire_text      = fileread(filename_string);
3      total_num_characters = numel(entire_text);
4      p1               = 1;
5      p2               = 1;
6      currententry     = "";
7      currline         = [];
8      finalmatrix      = [];
9
10     while(p2 <= total_num_characters)
11
12         curr_char = entire_text(p2);
13
14         if(curr_char == ',' || curr_char == "\n")
15             curr_entry = entire_text(p1:p2-1);
16             currline   = [currline, str2num(curr_entry)];
17
18             if (curr_char == "\n")
19                 finalmatrix = [finalmatrix; currline];
20                 currline    = [];
21             end
22
23             p1 = p2 + 1;
24         end
25         p2 += 1;
26     end
27 end

```