Autonomous Underwater Vehicle: A Surveillance Protocol

S.V. Rajendran

September 26, 2025

Preface

This project is an attempt at combining all of my major skills into creating a simulation, imaging, perception and control pipeline for Autonomous Underwater Vehicles (AUV). As such, creating this project involves creating a number of pipelines.

The first pipeline is the signal simulation pipeline. The signal simulation pipeline involves sea-floor point-cloud creation and simulating the signals received by the sensor arrays of the AUV. The signals recorded by the sensor-arrays on the AUV contains information from the surrounding environment. The imaging pipeline performs certain operations on the recorded signals to obtain acoustic images of the surrounding environment. To that end, this pipeline involves the topics of signal processing, linear algebra, signals and systems.

As such, the second pipeline is the imaging pipeline. The inputs to the imaging pipeline is the signals recorded by the different sensor-arrays of the AUV, in addition to the parameters of the AUV and its components. This pipeline involves match-filtering, focussing and beamforming operations to create acoustic images of the surrounding environment. Depending on the number of ULAs present, the imaging pipeline is responsible for creating multiple acoustic images in real-time. Thus, this pipeline involves the topics of Digital Signal Processing, Match-Filtering, Estimation and Detection Theory and so on.

The images created by the imaging pipeline are fed to the perception-to-control pipeline. This pipeline takes in the image formed created from the ULA signals, parameters of AUV and its components, and some historical data, it provides instructions regarding the movement of the AUV. The mapping from the inputs to the controls is called policy. Learning policies is a core part of reinforcement learning. Thus, this pipeline mainly involves the topics of reinforcement learning. And since we'll be using convolutional neural nets and transformers for learning the policies, this pipeline involves a significant amount of machine and deep learning.

The final result is an AUV that is primarily trained to map an area of the sea-floor in a constant surveillance mode. The RL-trained policy will also be trained to deal with different kinds of sea-floor terrains: those containing hills, valleys, and path-obstructing features. Due to the resource constrained nature of the marine vessel, we also prioritize efficient policies in the policy-training pipeline.

The project is currently written in C++. And since there is non-trivial amount of training and adaptive features in the pipelines, we'll be using LibTorch (the C++ API of PyTorch) to enable computation graphs, backpropagation and thereby, learning in our AUV pipeline. However, for the sections where a computation graph is not required, such as signal simulation, we will be writing templated STL code.

Introduction

Contents

Chapter 1

Setup

1.1 Overview

- Clone the AUV repository: `https://github.com/vrsreeganesh/AUV.git`.
- This can be performed by entering the terminal, "cd"-ing to the directory you wish and then typing: `git clone https://github.com/vrsreeganesh/AUV.git` and press enter.
- Note that in case it has not been setup, ensure github setup in the terminal. If not familiar with the whole git work-routine, I suggest sticking to Github Desktop. Its a lot easier and the best to get started right away.
- Or if you do not wish to follow a source-control approach, just download the repository as a zip file after clicking the blue code button.

Chapter 2

Underwater Environment Setup

Overview

All physical matter in this framework is represented using point-clouds. Thus, the sea-floor also is represented using a number of 3D points. In addition to the coordinates, the points also have the additional property of "reflectivity". It is the impulse response of that point.

Sea-floors in real-life are rarely flat. They often contain valleys, mountains, hills and much richer geographical features. Thus, training an agent to function in such environments call for the creation of similar structures in our simulations. Even though there must be infinite variations in the structures found under water, we shall take a constrained and structured approach to creating these variations. To that end, we shall start with an additive approach. We define few types of underwater structure whos shape, size and what not can be parameterized to enable creation of random seafloors. The full-script for creating the sea-floor is available in section **??**.

2.1 Sea "Floor"

The first entity that we will be adding to create the seafloor is the floor itself. This is set of points that are in the lowest ring of point-clouds in the point-cloud representation of the total sea-floor.

The most basic approach to creating this is to create a flat seafloor, where all the points have the same height. While this is a good place to start, it is good to bring in some realism to the seafloor. To that end, we shall have some rolling hills as the sea-floor. Each "hill " is created using the method outlined in Algorithm **??**. The method involves deciding the location of the hills, the dimension of the hills and then designing a hill by combining an exponential function and a cosine function. We're aiming to essentially produce gaussian-looking sea-floor hills. After the creation, this becomes the set of points representing the lowest set of points in the overall seafloor structure.

Algorithm 1 Hill Creation

Input: Mean vector $\mathbf{m}$, Dimension vector $\mathbf{d}$, 2D points $\mathbf{P}$
Output: Updated $\mathbf{P}$ with hill heights
num_hills $\leftarrow$ numel$(\mathbf{m}_x)$
$H \leftarrow$ Zeros tensor of size $(1, \text{numel}(\mathbf{P}_x))$
for $i = 1$ to num_hills **do**
    $x_{\text{norm}} \leftarrow \frac{\frac{\pi}{2}(\mathbf{P}x - \mathbf{m}x[i])}{\mathbf{d}x[i]}$
    $y_{\text{norm}} \leftarrow \frac{\frac{\pi}{2}(\mathbf{P}y - \mathbf{m}y[i])}{\mathbf{d}y[i]}$
    $h_x \leftarrow \cos(x_{\text{norm}}) \cdot e^{\frac{|x_{\text{norm}}|}{10}}$
    $h_y \leftarrow \cos(y_{\text{norm}}) \cdot e^{\frac{|y_{\text{norm}}|}{10}}$
    $h \leftarrow \mathbf{d}_z[i] \cdot h_x \cdot h_y$
    Apply boundary conditions:
    if $x_{\text{norm}} > \frac{\pi}{2}$ or $x_{\text{norm}} < -\frac{\pi}{2}$ or $y_{\text{norm}} > \frac{\pi}{2}$ or $y_{\text{norm}} < -\frac{\pi}{2}$ **then**
        $h \leftarrow 0$
    end if
    $H \leftarrow H + h$
end for
$\mathbf{P} \leftarrow$ concatenate$([\mathbf{P}, H])$

2.2 Simple Structures

2.2.1 Boxes

These are apartment like structures that represent different kinds of rectangular pyramids. These don't necessarily correspond to any real-life structures but these are super simple structures that will help us assess the shadows that are created in the beamformed acoustic image.

Algorithm 2 Generate Box Meshes on Sea Floor

Require: $across\_track\_length$, $along\_track\_length$, $box\_coordinates$, $box\_reflectivity$

Initialize min/max width, length, height, meshdensity, reflectivity, and number of boxes
Generate random center points for boxes:
$midxypoints \leftarrow \text{rand}([3, num\_boxes])$
$midxypoints[0] \leftarrow midxypoints[0] \times across\_track\_length$
$midxypoints[1] \leftarrow midxypoints[1] \times along\_track\_length$
$midxypoints[2] \leftarrow 0$
Assign random dimensions to each box:
$boxwidths \leftarrow \text{rand}(num\_boxes) \times (max\_width - min\_width) + min\_width$
$boxlengths \leftarrow \text{rand}(num\_boxes) \times (max\_length - min\_length) + min\_length$
$boxheights \leftarrow \text{rand}(num\_boxes) \times (max\_height - min\_height) + min\_height$
for $i = 1$ to num_boxes **do**
    Generate mesh points along each axis:
    $xpoints \leftarrow \text{linspace}(-boxwidths[i]/2, boxwidths[i]/2, boxwidths[i] \times meshdensity)$
    $ypoints \leftarrow \text{linspace}(-boxlengths[i]/2, boxlengths[i]/2, boxlengths[i] \times meshdensity)$
    $zpoints \leftarrow \text{linspace}(0, boxheights[i], boxheights[i] \times meshdensity)$
    Generate 3D mesh grid:
    $X, Y, Z \leftarrow \text{meshgrid}(xpoints, ypoints, zpoints)$
    Reshape X, Y, Z into 1D tensors
    Compute final coordinates:
    $boxcoordinates \leftarrow \text{cat}(X, Y, Z)$
    $boxcoordinates[0] \leftarrow boxcoordinates[0] + midxypoints[0][i]$
    $boxcoordinates[1] \leftarrow boxcoordinates[1] + midxypoints[1][i]$
    $boxcoordinates[2] \leftarrow boxcoordinates[2] + midxypoints[2][i]$
    Generate reflectivity values:
    $boxreflectivity \leftarrow meshreflectivity + \text{rand}(1, \text{size}(boxcoordinates)) - 0.5$
    Append data to final tensors:
    $box\_coordinates \leftarrow \text{cat}(box\_coordinates, boxcoordinates, 1)$
    $box\_reflectivity \leftarrow \text{cat}(box\_reflectivity, boxreflectivity, 1)$
end for

2.2.2 Sphere

Just like boxes, these are structures that don't necessarily exist in real life. We use this to essentially assess the shadowing in the beamformed acoustic image.

Algorithm 3 Sphere Creation

num_hills ← Number of Hills

Chapter 3

Hardware Setup

Overview

The AUV contains a number of hardware that enables its functioning. A real AUV contains enough components to make a victorian child faint. And simulating the whole thing and building pipelines to model their working is not the kind of project to be handled by a single engineer. So we'll only model and simulate those components that are absolutely required for the running of these pipelines.

3.1 Transmitter

Probing systems are those systems that send out a signal, listen to the reflection and infer qualitative and quantitative qualities of the environment, matter or object, it was trying to infer information about. The transmitter is one of the most fundamental components of probing systems. As the name suggests, the transmitter is the equipment responsible for sending out the probing signal into the medium.

Transmitters are of many kinds. But the ones that we will be considering will be directed transmitters, which means that these transmitters have an associated beampattern. To the uninitiated, this means that the power of the transmitted signal is not transmitted in all directions equally. A beampattern is a graphical representation of the power received by an ideal receiver when placed at different angles.

Transmitters made out of a linear-array of individual transmitters use beamforming to "direct" the major power of the transmitter. These kind of systems have well studied beampatterns which can be utilized in our simulations. These kind of studies and inculcating that in our pipelines produce accurate signal simulation pipelines.

For now, we stick to a very simple model of a transmitter. We assume that the transmitter sends out the power equally into a particular cone from the AUV position.

The full-script for the setup of the transmitter is given in section **??** and the class definition for the transmitter is given in section **??**.

3.2 Uniform Linear Array

Perhaps the most important component of probing systems are the "listening" systems. After "illuminating" the medium with the signal, we need to listen to the reflections in order to infer properties. In fact, there are some probing systems that do not use transmitter. Thus, this easily makes the case for the simple fact that the "listening" components of probing systems are the most important components of the whole system.

Uniform arrays are of many kinds but the most popular ones are uniform linear arrays and uniform planar arrays. The arrays in this case contain a number of sensors arranged in a uniform manner across a line or a plane.

Linear arrays have the property that the information obtained from elevation, ϕ is no longer available due to the dimensionality of the array-structure. Thus, the images obtained from processing the signals recorded by a uniform linear array will only have two-dimensions: the azimuth, θ and the range, r.

Thus, for 3D imaging, we shall be working with planar arrays. However, due to the higher dimensionality of the output signal, the class of algorithms required to create 3D images are a lot more computationally efficient. In addition, due to the simpler nature of the protocols involved with our AUV, uniform linear arrays will work just fine.

3.3 Marine Vessel

"Marine Vessel" refers to the platform on which the previously mentioned components are mounted on. These usually range from ships to submarines to AUVs. In our context, since we're working with the AUV, the marine vessel in our case is the AUV.

The standard AUV has four degrees of freedom. Unlike drones that has practically all six degrees of freedom, AUV's are two degrees short. However, that is okay for the functionalities most drones are designed for. But for now, we're allowing the simulation to create a drone that has all six degrees of freedom. This will soon be patched.

Chapter 4

Signal Simulation

Overview

- Define LFM.
- Define shadowing.
- Simulate Signals (basic)
- Simulate Signals with additional effects (doppler)

4.1 Transmitted Signal

- In probing systems, which are systems which transmit a signal and infer qualitative and quantiative characterisitics of the environment from the signal return, the ideal signal is the dirac delta signal. However, dirac-deltas are nearly impossible to create because of their infinite bandwidth structure. Thus, we need to use something else that is more practical but at the same time, gets us quite close the diract-delta. So we use something of a watered-down delta-function, which is a bandlimited delta function, or the linear frequency-modulated signal. The LFM is a asignal whose frequency increases linearly in its duration. This means that the signal has a flat magnitude spectrum but quadratic phase.
- The LFM is characterised by the bandwidth and the center-frequency. The higher the resolution required, the higher the transmitted bandwidth is. So bandwidth is a characterizing factor. The higher the bandwidth, the better the resolution obtained.
- The transmitted signals used in these cases depend highly on the kind of SONAR we're using it for. The systems we're using currently contain one FLS and two side-scan or 3 FLS (I'm yet to make up mind here).
- The signal is defined in setup-script of the transmitter. Please refer to section: **??**;

4.2 Signal Simulation

1. The signals simulation is performed using simple ray-tracing. The distance travelled from the transmitted to scatterer and then the sensor is calculated for each scatter-sensor pair. And the transmitted signal is placed at the recording of each sensor corresponding to each scatterer.
2. First we obtain the set of scatterers that reflect the transmitted signal.
3. The distance between all the sensors and the scatterer distances are calculated.
4. The time of flight from the transmitter to each scatterer and each sensor is calculated.
5. This time is then calculated into sample number by multiplying with the sampling-frequency of the uniform linear arrays.
6. We then build a signal matrix that has the dimensions corresponding to the number of samples that are recorded and the number of sensors that are present in the sensor-array.
7. We place impulses in the points corresponding to when the signals arrives from the scatterers. The result is a matrix that has x-dimension as the number of samples and the y-dimension as the number of sensors.
8. Each column is then convolved (linearly convolved) with the transmitted signal. The resulting matrix gives us the signal received by each sensor. Note that this method doesn't consider doppler effects. This will be added later.

Algorithm 4 Signal Simulation

ScatterCoordinates ←
ScatterReflectivity ←
AngleDensity ← Quantization of angles per degree.
AzimuthalBeamwidth ← Azimuthal Beamwidth
RangeCellWidth ← The range-cell width

4.3 Ray Tracing

- There are multiple ways for ray-tracing.
- The method implemented during the FBLS and SS SONARs weren't super efficient as it involved pair-wise dot-products. Which becomes an issue when the number of points are increased, which is the case when the range is super high or the beamwidth is super high.

4.3.1 Pairwise Dot-Product

- In this method, given the coordinates of all points that are currently in the illumination cone, we find the cosines between every possible pairs of points.
- This is where the computational complexity arises as the number of dot products increase exponentially with increasing number of points.

- This method is a liability when it comes to situations where the range is super high or when the angle-beamwidth is non-narrow.

4.3.2 Range Histogram Method

- Given the angular beamwidths: azimuthal beamwidth and elevation beamwidth, we quantize square cone into a number of different values (note that the square cone is not an issue as the step before ensures conical subsetting).
- We split the points into different ”range-cells”.
- For each range-cell, we make a 2D histogram of azimuths and elevations. Then within each range-cell and for each azimuth-elevation pair, we find the closest point and add it to the check-box.
- In the next range-cell, we only work with those azimuth-elevation pairs whose check-box has not been filled. Since, for the filled ones, the filled scatter will shadow the othersin the following range cells.

Algorithm 5 Range Histogram Method

ScatterCoordinates ←
ScatterReflectivity ←
AngleDensity ← Quantization of angles per degree.
AzimuthalBeamwidth ← Azimuthal Beamwidth
RangeCellWidth ← The range-cell width

Chapter 5

Imaging

Overview

- Present basebanding, low-pass filtering and decimation.
- Present beamforming.
- Present different synthetic-aperture concepts.

5.1 Decimation

1. Due to the large sampling-frequencies employed in imaging SONAR, it is quite often the case that the amount of samples received for just a couple of milliseconds make for non-trivial data-size.
2. In such cases, we use some smart signal processing to reduce the data-size without loss of information. This is done using the fact that the transmitted signal is non-baseband. THis means that using a method known as quadrature modulation, we can maintain the information content without the humongous amount data.
3. After basebanding the signal, this process involves decimation of the signal respecting the bandwidth of the transmitted signal.

5.1.1 Basebanding

1. Basebanding is performed utilizing the frequency-shifting property of the fourier transform

$$x(t)e^{j2\pi\omega_0 t} \leftrightarrow X(\omega - \omega_0)$$

2. Since we're working with digital signals, this is implemented in the following manner

$$x[n]e^{j\frac{2\pi k_0 n}{N}} \leftrightarrow X(k - k_0)$$

Algorithm 6 Basebanding

ScatterCoordinates ←
ScatterReflectivity ←
AngleDensity ← Quantization of angles per degree.
AzimuthalBeamwidth ← Azimuthal Beamwidth
RangeCellWidth ← The range-cell width

5.1.2 Lowpass filtering

1. Now that we have the signal in the baseband, we lowpass filter the signal based on the bandwidth of the signal. Since we're perfectly centering the signal using f_c, we can have the cutoff-frequency of the lowpass filter to be just above half the bandwidth of the transmitted signal. Note that the signals should not be brought down back into the real-domain using abs() or real() functions since the negative frequencies are no longer symmetrical.

2. After low-pass filtering, we have a band-restricted signal that contains all of the data in the baseband. This allows for decimation, which is what we'll do in the next step.

5.1.3 Decimation

1. Now that we have the bandlimited signal, what we shall do is decimation. Decimation essentially involves just taking every n-th sample where n in this case is the decimation factor.

2. The resulting signal contains the same information as that of the real-sampled signal but with much less number of samples.

5.2 Match-Filtering

1. To understand why match-filtering is going on, it is important to understand pulse compression.

2. In "probing" systems, which are basically systems where we send out some signal, listen to the reflection and infer quantitative and qualitative aspects of the environment, the best signal is the impulse signal (see Dirac Delta). However. this signal is not practical to use. Primarily due to the very simple fact that this particular signal has a flat and infinite bandwidth. However, this signal is the idea.

3. So instead, we're left with using signals that have a finite length, $T_{\text{Transmitted Signal}}$. However, the issue with that is that a scatter of initesimal dimension produce a response that has a length of $T_{\text{Transmitted Signal}}$. Thus, it is important to ensure that the response of each object, scatter or what not has comparable dimensions. This is where pulse compression comes in. Using this technique, we transform the received signal to produce a signal that is as close as possible to the signal we'd receive if we were to send out a diract delta pulse.

4. Thus, this process involves something of a detection. The closest method is something of a correlation filter where we run a copy of the transmitted signal through the received recording and take inner-products at each time step (known as the cor-

relation operation). This method works great if we're in the real domain. However, thanks to the quadrature demodulation we performed, this process is now no longer valid. But the idea remainds the same. The point of doing a correlation analysis is so that where there is a signal, a spike appears. The sample principle is used to develop the match-filter.

5. We want to produce a filter, which when convolved with the received signal produces a spike. Since we're trying to produce something similar to the response of an ideal transmission system, we want the output to be that of an ideal spike, which is the delta function. So we're essentially trying to find a filter, which when multiplied with the transmitted signal, produces the diract delta.

6. The answer can be found by analyzing the frequency domain. The frequency domain basis representation of the delta-function is a flat magnitude and linear phase. Thus, this means that the filter that we use on the transmitted signal must produce a flat magnitude and linear phase. The transmitted signal that we're working with, being an LFM, means that the magnitude is already flat. The phase, however, is quadratic. So we need the matched filter to have a flat magnitude and a quadratic phase that cancels away that of the transmitted signa's quadratic component. All this leads to the best candidate: the complex conjugate of the transmitted signal. However, since we're now working with the quadrature demodulated signal, the matched filter is the complex conjugate of the quadrature demodulated transmitted signal.

7. So once the filter is made, convolving that with the received signal produces a number of spikes in the processed signal. Note that due to working in the digital domain and some other factors, the spikes will not be perfect. Thus it is not safe to take the abs() or real() just yet. We'll do that after beamforming.

8. But so far, this marks the first step of the perception pipeline.

Algorithm 7 Match-Filtering

ScatterCoordinates ←
ScatterReflectivity ←
AngleDensity ← Quantization of angles per degree.
AzimuthalBeamwidth ← Azimuthal Beamwidth
RangeCellWidth ← The range-cell width

Beamforming

- Prior to imaging, we precompute the range-cell characteristics.
- In addition, we also calculate the delays given to each sensor for each of those range-azimuth combinations.
- Those are then stored as a look-up table member of the class.
- At each-time step, what we do is we buffer split the simulated/received signal into a 3D matrix, where each signal frame corresponds to the signals for a particular range-cell.
- Then for each range-cell, we beamform using the delays we precalculated. We perform this without loops in order to utilize CPU and reduce latency.

Algorithm 8 Beamforming

ScatterCoordinates ←
ScatterReflectivity ←
AngleDensity ← Quantization of angles per degree.
AzimuthalBeamwidth ← Azimuthal Beamwidth
RangeCellWidth ← The range-cell width

Chapter 6

Control Pipeline

Overview

1. The inputs to the control-pipeline is the images obtained from previous pipeline.
2. Currently the plan is to use DQN.

DQN

1. Here we're essentially trying to create a control pipeline that performs the protocol that we need.
2. The aim of the AUV is to continuously map a particular area of the sea-floor and perform it despite the presence of sea-floor structures.
3.

Algorithm 9 DQN

ScatterCoordinates ←
ScatterReflectivity ←
AngleDensity ← Quantization of angles per degree.
AzimuthalBeamwidth ← Azimuthal Beamwidth
RangeCellWidth ← The range-cell width

Artificial Acoustic Imaging

1. In order to ensure faster development, we shall start off with training the DQN algorithm with artificial acoustic images. This is rather important due to the fact that the imaging pipelines (currently) has some non-trivial latency. This means that using those pipelines to create the inputs to the DQN algorithm will skyrocket the training time.
2. So the approach that we shall be taking will be write functions to create artifical acoustic images directly from the scatterer-coordinates and scatterer-reflectivity values. The latency for these functions are negligible compared to that of beamforming-

based imaging algorithms. The function for this has been added and is available in section **??** under the function name, *nfdc_createAcousticImage*. Please note that these functions are not to be directly called from the main function. Instead, it is expected that the main function calls the AUV classes's method, *create_ArtificialAcousticImage*. This function calls the class ULA's method appropriately.

3. After the ULA's create their respective acoustic images, they are put together, either by dimension-wise concatenation or depth-wise concatenation and feed to the neural net to produce control sequences.

4. We need to work on the dimensions of these images though. The best thing to do right now is to finalize the transmitter and receiver parameters and then over-estimate the dimensions of the final beamforming-produced image. We shall then use these dimensions to create the artificial acoustic image and start training the policy.

Algorithm 10 Artifical Acoustic Imaging

ScatterCoordinates ← Coordinates of points in the point-cloud.
auvCoordinates ← Coordinates of AUV/ULA.

Chapter 7

Results

Chapter 8

Software

Overview

-

8.1 Class Definitions

8.1.1 Class: Scatter

The following is the class definition used to encapsulate attributes and methods of the scatterers.

```
// // header-files
// #include <iostream>
// #include <ostream>
// #include <torch/torch.h>

// #pragma once

// // hash defines
// #ifndef PRINTSPACE
// #define PRINTSPACE   std::cout<<"\n\n\n\n\n\n\n\n"<<std::endl;
// #endif
// #ifndef PRINTSMALLLINE
// #define PRINTSMALLLINE std::cout<<"------------------------------------------------"<<std::endl;
// #endif
// #ifndef PRINTLINE
// #define PRINTLINE    std::cout<<"================================================"<<std::endl;
// #endif
// #ifndef DEVICE
//    #define DEVICE       torch::kMPS
//    // #define DEVICE       torch::kCPU
// #endif


// #define PI           3.14159265


// // function to print tensor size
// void print_tensor_size(const torch::Tensor& inputTensor) {
//    // Printing size
//    std::cout << "[";
//    for (const auto& size : inputTensor.sizes()) {
//        std::cout << size << ",";
//    }
//    std::cout << "\b]" <<std::endl;
```

```
// }

// // Scatterer Class = Scatterer Class
// // Scatterer Class = Scatterer Class
// // Scatterer Class = Scatterer Class
// // Scatterer Class = Scatterer Class
// // Scatterer Class = Scatterer Class
// class ScattererClass{
// public:

//     // public variables
//     torch::Tensor coordinates; // tensor holding coordinates [3, x]
//     torch::Tensor reflectivity; // tensor holding reflectivity [1, x]

//     // constructor = constructor
//     ScattererClass(torch::Tensor arg_coordinates = torch::zeros({3,1}),
//                    torch::Tensor arg_reflectivity = torch::zeros({3,1})):
//                    coordinates(arg_coordinates),
//                    reflectivity(arg_reflectivity) {}

//     // overloading output
//     friend std::ostream& operator<<(std::ostream& os, ScattererClass& scatterer){

//         // printing coordinate shape
//         os<<"\t> scatterer.coordinates.shape = ";
//         print_tensor_size(scatterer.coordinates);

//         // printing reflectivity shape
//         os<<"\t> scatterer.reflectivity.shape = ";
//         print_tensor_size(scatterer.reflectivity);

//         // returning os
//         return os;
//     }

//     // copy constructor from a pointer
//     ScattererClass(ScattererClass* scatterers){

//         // copying the values
//         this->coordinates = scatterers->coordinates;
//         this->reflectivity = scatterers->reflectivity;
//     }

// };

template <typename T>
class ScattererClass
{
public:
    // members
    std::vector<std::vector<T>> coordinates;
    std::vector<T>           reflectivity;

    // Constructor
    ScattererClass()  {}

    // Constructor
    ScattererClass(std::vector<std::vector<T>> coordinates_arg,
                   std::vector<T>           reflectivity_arg):
                   coordinates(coordinates_arg),
                   reflectivity(reflectivity_arg) {}

    // Save to CSV
    void savetocsv(){
        fWriteMatrix(this->coordinates, "../csv-files/coordinates.csv");
        fWriteVector(this->reflectivity, "../csv-files/reflectivity.csv");
    }
};
```

8.1.2 Class: Transmitter

The following is the class definition used to encapsulate attributes and methods of the projectors used.

8.1.3 Class: Uniform Linear Array

The following is the class definition used to encapsulate attributes and methods for the uniform linear array.

```
// // bringing in the header files
// #include <cstdint>
// #include <iostream>
// #include <ostream>
// #include <stdexcept>
// #include <torch/torch.h>
// #include <omp.h>         // the openMP


// // class definitions
// #include "ScattererClass.h"
// #include "TransmitterClass.h"

// // bringing in the functions
// #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fPrintTensorSize.cpp"
// #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fConvolveColumns.cpp"
// #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fBuffer2D.cpp"
// #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fConvolve1D.cpp"
// #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fCart2Sph.cpp"

// #pragma once

// // hash defines
// #ifndef PRINTSPACE
//     #define PRINTSPACE    std::cout<<"\n\n\n\n\n\n\n\n"<<std::endl;
// #endif
// #ifndef PRINTSMALLLINE
//     #define PRINTSMALLLINE
    std::cout<<"--------------------------------------------------------------------------------"<<std::endl;
// #endif
// #ifndef PRINTLINE
//     #define PRINTLINE
    std::cout<<"================================================================================"<<std::endl;
// #endif
// #ifndef PRINTDOTS
//     #define PRINTDOTS
    std::cout<<"................................................................................"<<std::endl;
// #endif


// #ifndef DEVICE
//     // #define DEVICE       torch::kMPS
//     #define DEVICE       torch::kCPU
// #endif

// #define PI           3.14159265
// #define COMPLEX_1j          torch::complex(torch::zeros({1}), torch::ones({1}))

// // #define DEBUG_ULA true
// #define DEBUG_ULA false



// class ULAClass{
// public:
//     // intrinsic parameters
//     int num_sensors;                // number of sensors
//     float inter_element_spacing;    // space between sensors
//     torch::Tensor coordinates;      // coordinates of each sensor
//     float sampling_frequency;       // sampling frequency of the sensors
//     float recording_period;         // recording period of the ULA
//     torch::Tensor location;         // location of first coordinate

//     // derived stuff
//     torch::Tensor sensorDirection;
//     torch::Tensor signalMatrix;

//     // decimation-related
```

```
66  //    int decimation_factor;                          // the new decimation factor
67  //    float post_decimation_sampling_frequency;       // the new sampling frequency
68  //    torch::Tensor lowpassFilterCoefficientsForDecimation; //
69
70  //    // imaging related
71  //    float range_resolution;         // theoretical range-resolution = $\frac{c}{2B}$
72  //    float azimuthal_resolution;     // theoretical azimuth-resolution =
        $\frac{\lambda}{(N-1)*inter-element-distance}$
73  //    float range_cell_size;          // the range-cell quanta we're choosing for efficiency trade-off
74  //    float azimuth_cell_size;        // the azimuth quanta we're choosing
75  //    torch::Tensor mulFFTMatrix;     // the matrix containing the delays for each-element as a slot
76  //    torch::Tensor azimuth_centers;  // tensor containing the azimuth centeres
77  //    torch::Tensor range_centers;    // tensor containing the range-centers
78  //    int frame_size;                 // the frame-size corresponding to a range cell in a decimated signal
        matrix
79  //    torch::Tensor matchFilter;      // torch tensor containing the match-filter
80  //    int num_buffer_zeros_per_frame; // number of zeros we're adding per frame to ensure no-rotation
81  //    torch::Tensor beamformedImage;  // the beamformed image
82  //    torch::Tensor cartesianImage;
83
84  //    // artificial acoustic-image related
85  //    torch::Tensor currentArtificalAcousticImage; // the acoustic image directly produced
86
87  //    // constructor
88  //    ULAClass(int numsensors             = 32,
89  //             float inter_element_spacing = 1e-3,
90  //             torch::Tensor coordinates   = torch::zeros({3, 2}),
91  //             float sampling_frequency    = 48e3,
92  //             float recording_period      = 1,
93  //             torch::Tensor location      = torch::zeros({3,1}),
94  //             torch::Tensor signalMatrix  = torch::zeros({1, 32}),
95  //             torch::Tensor lowpassFilterCoefficientsForDecimation = torch::zeros({1,10})):
96  //             num_sensors(numsensors),
97  //             inter_element_spacing(inter_element_spacing),
98  //             coordinates(coordinates),
99  //             sampling_frequency(sampling_frequency),
100 //             recording_period(recording_period),
101 //             location(location),
102 //             signalMatrix(signalMatrix),
103 //             lowpassFilterCoefficientsForDecimation(lowpassFilterCoefficientsForDecimation){
104 //                // calculating ULA direction
105 //                torch::Tensor sensorDirection = coordinates.slice(1, 0, 1) - coordinates.slice(1, 1, 2);
106
107 //                // normalizing
108 //                float normvalue = torch::linalg_norm(sensorDirection, 2, 0, true,
        torch::kFloat).item<float>();
109
110
111 //                if (normvalue != 0){
112 //                    sensorDirection = sensorDirection / normvalue;
113 //                }
114
115 //                // copying direction
116 //                this->sensorDirection = sensorDirection.to(DATATYPE);
117 //        }
118
119 //    // overrinding printing
120 //    friend std::ostream& operator<<(std::ostream& os, ULAClass& ula){
121 //        os<<"\t number of sensors : "<<ula.num_sensors        <<std::endl;
122 //        os<<"\t inter-element spacing: "<<ula.inter_element_spacing <<std::endl;
123 //        os<<"\t sensor-direction "   <<torch::transpose(ula.sensorDirection, 0, 1)<<std::endl;
124 //        PRINTSMALLLINE
125 //        return os;
126 //    }
127
128 //    /* ========================================================================
129 //    Aim: Init
130 //    ------------------------------------------------------------------------ */
131 //    void init(TransmitterClass* transmitterObj){
132
133 //        // calculating range-related parameters
134 //        this->range_resolution    = 1500/(2 * transmitterObj->fc);
135 //        this->range_cell_size     = 40 * this->range_resolution;
136 //        if (DEBUG_ULA) std::cout << "\t ULACLASS::init: line 136" << std::endl;
137
```

```
138 //        // status printing
139 //        if (DEBUG_ULA) {
140 //            std::cout << "\t\t ULAClass::init():: this->range_resolution = " \
141 //                      << this->range_resolution                              \
142 //                      << std::endl;
143 //            std::cout << "\t\t ULAClass::init():: this->range_cell_size = " \
144 //                      << this->range_cell_size                               \
145 //                      << std::endl;
146 //        }
147 //        if (DEBUG_ULA) std::cout << "\t ULACLASS::init: line 147" << std::endl;
148 
149 //        // calculating azimuth-related parameters
150 //        this->azimuthal_resolution =                                         \
151 //            (1500/transmitterObj->fc)                                        \
152 //            /((this->num_sensors-1)*this->inter_element_spacing);
153 //        this->azimuth_cell_size      = 2 * this->azimuthal_resolution;
154 //        if (DEBUG_ULA) std::cout << "\t ULACLASS::init: line 154" << std::endl;
155 
156 //        // creating and storing the match-filter
157 //        this->nfdc_CreateMatchFilter(transmitterObj);
158 //        if (DEBUG_ULA) std::cout << "\t ULACLASS::init: line 158" << std::endl;
159 //    }
160 
161 //    // Create match-filter
162 //    void nfdc_CreateMatchFilter(TransmitterClass* transmitterObj){
163 
164 //        // creating matrix for basebanding the signal
165 //        torch::Tensor basebanding_vector =             \
166 //            torch::linspace(                           \
167 //                0,                                     \
168 //                transmitterObj->Signal.numel()-1,      \
169 //                transmitterObj->Signal.numel()         \
170 //                ).reshape(transmitterObj->Signal.sizes());
171 //        basebanding_vector =                               \
172 //            torch::exp(                                    \
173 //                -1 * COMPLEX_1j * 2 * PI                       \
174 //                * (transmitterObj->fc/this->sampling_frequency) \
175 //                * basebanding_vector);
176 //        if (DEBUG_ULA) std::cout << "\t\t ULAClass::nfdc_createMatchFilter: line 176" << std::endl;
177 
178 //        // multiplying the signal with the basebanding vector
179 //        torch::Tensor match_filter =                   \
180 //            torch::mul(transmitterObj->Signal,         \
181 //                       basebanding_vector);
182 //        if (DEBUG_ULA) std::cout << "\t\t ULAClass::nfdc_createMatchFilter: line 182" << std::endl;
183 
184 //        // low-pass filtering to get the baseband signal
185 //        fConvolve1D(match_filter, this->lowpassFilterCoefficientsForDecimation);
186 //        if (DEBUG_ULA) std::cout << "\t\t ULAClass::nfdc_createMatchFilter: line 186" << std::endl;
187 
188 //        // creating sampling-indices
189 //        int decimation_factor = \
190 //            std::floor((static_cast<float>(this->sampling_frequency)/2) \
191 //                       /(static_cast<float>(transmitterObj->bandwidth)/2));
192 //        int final_num_samples = \
193 //            std::ceil(static_cast<float>(match_filter.numel())/static_cast<float>(decimation_factor));
194 //        torch::Tensor sampling_indices = \
195 //            torch::linspace(1, \
196 //                            (final_num_samples-1) * decimation_factor,
197 //                            final_num_samples).to(torch::kInt) - torch::tensor({1}).to(torch::kInt);
198 //        if (DEBUG_ULA) std::cout << "ULAClass::nfdc_createMatchFilter: line 197" << std::endl;
199 
200 //        // sampling the signal
201 //        match_filter = match_filter.index({sampling_indices});
202 
203 //        // taking conjugate and flipping the signal
204 //        match_filter = torch::flipud( match_filter);
205 //        match_filter = torch::conj(  match_filter);
206 
207 //        // storing the match-filter to the class member
208 //        this->matchFilter = match_filter;
209 //    }
210 
211 //    // overloading the "=" operator
212 //    ULAClass& operator=(const ULAClass& other){
```

```
213 //        // checking if copying to the same object
214 //        if(this == &other){
215 //            return *this;
216 //        }
217 
218 //        // copying everything
219 //        this->num_sensors        = other.num_sensors;
220 //        this->inter_element_spacing = other.inter_element_spacing;
221 //        this->coordinates        = other.coordinates.clone();
222 //        this->sampling_frequency = other.sampling_frequency;
223 //        this->recording_period   = other.recording_period;
224 //        this->sensorDirection    = other.sensorDirection.clone();
225 
226 //        // new additions
227 //        // this->location           = other.location;
228 //        this->lowpassFilterCoefficientsForDecimation = other.lowpassFilterCoefficientsForDecimation;
229 //        // this->sensorDirection    = other.sensorDirection.clone();
230 //        // this->signalMatrix       = other.signalMatrix.clone();
231 
232 
233 //        // returning
234 //        return *this;
235 //    }
236 
237 //    // build sensor-coordinates based on location
238 //    void buildCoordinatesBasedOnLocation(){
239 
240 //        // length-normalize the sensor-direction
241 //        this->sensorDirection = torch::div(this->sensorDirection, torch::linalg_norm(this->sensorDirection,
    \
242 //                                                                                      2, 0, true, \
243 //                                                                                      DATATYPE));
244 //        if(DEBUG_ULA) std::cout<<"\t ULAClass: line 105 \n";
245 
246 //        // multiply with inter-element distance
247 //        this->sensorDirection = this->sensorDirection * this->inter_element_spacing;
248 //        this->sensorDirection = this->sensorDirection.reshape({this->sensorDirection.numel(), 1});
249 //        if(DEBUG_ULA) std::cout<<"\t ULAClass: line 110 \n";
250 
251 //        // create integer-array
252 //        // torch::Tensor integer_array = torch::linspace(0, \
253 //        //                                               this->num_sensors-1, \
254 //        //                                               this->num_sensors).reshape({1,
    this->num_sensors}).to(DATATYPE);
255 //        torch::Tensor integer_array = torch::linspace(0,                                    \
256 //                                                      this->num_sensors-1,                  \
257 //                                                      this->num_sensors).reshape({1,        \
258 //                                                                                  this->num_sensors});
259 //        std::cout<<"integer_array = "; fPrintTensorSize(integer_array);
260 //        if(DEBUG_ULA) std::cout<<"\t ULAClass: line 116 \n";
261 
262 //        //
263 //        torch::Tensor test = torch::mul(torch::tile(integer_array, {3, 1}).to(DATATYPE), \
264 //                                        torch::tile(this->sensorDirection, {1,
    this->num_sensors}).to(DATATYPE));
265 //        this->coordinates = this->location + test;
266 //        if(DEBUG_ULA) std::cout<<"\t ULAClass: line 120 \n";
267 
268 //    }
269 
270 //    // signal simulation for the current sensor-array
271 //    void nfdc_simulateSignal(ScattererClass* scatterers,
272 //                             TransmitterClass* transmitterObj){
273 
274 //        // creating signal matrix
275 //        int numsamples    = std::ceil((this->sampling_frequency * this->recording_period));
276 //        this->signalMatrix = torch::zeros({numsamples, this->num_sensors}).to(DATATYPE);
277 
278 
279 //        // getting shape of coordinates
280 //        std::vector<int64_t> scatterers_coordinates_shape = \
281 //            scatterers->coordinates.sizes().vec();
282 
283 //        // making a slot out of the coordinates
284 //        torch::Tensor slottedCoordinates =                                  \
```

```
285 //            torch::permute(scatterers->coordinates.reshape({
286 //                scatterers_coordinates_shape[0],                          \
287 //                scatterers_coordinates_shape[1],                          \
288 //                1}                                                        \
289 //                ), {2, 1, 0}).reshape({
290 //                    1,                                                    \
291 //                    (int)(scatterers->coordinates.numel()/3),             \
292 //                    3});
293 
294 
295 //        // repeating along the y-direction number of sensor times.
296 //        slottedCoordinates =
297 //            torch::tile(slottedCoordinates,                               \
298 //                        {this->num_sensors, 1, 1});
299 //        std::vector<int64_t> slottedCoordinates_shape =                   \
300 //            slottedCoordinates.sizes().vec();
301 
302 
303 //        // finding the shape of the sensor-coordinates
304 //        std::vector<int64_t> sensor_coordinates_shape = \
305 //            this->coordinates.sizes().vec();
306 
307 //        // creating a slot tensor out of the sensor-coordinates
308 //        torch::Tensor slottedSensors =                                    \
309 //            torch::permute(this->coordinates.reshape({
310 //                sensor_coordinates_shape[0],                              \
311 //                sensor_coordinates_shape[1],                              \
312 //                1}), {2, 1, 0}).reshape({(int)(this->coordinates.numel()/3), \
313 //                                    1,                                    \
314 //                                    3});
315 
316 
317 //        // repeating slices along the x-coordinates
318 //        slottedSensors =                                                  \
319 //            torch::tile(slottedSensors,                                   \
320 //                        {1, slottedCoordinates_shape[1], 1});
321 
322 //        // slotting the coordinate of the transmitter and duplicating along dimensions [0] and [1]
323 //        torch::Tensor slotted_location =                                  \
324 //            torch::permute(this->location.reshape({3, 1, 1}),             \
325 //                           {2, 1, 0}).reshape({1,1,3});
326 //        slotted_location =                                                \
327 //            torch::tile(slotted_location, {slottedCoordinates_shape[0],   \
328 //                                           slottedCoordinates_shape[1],   \
329 //                                           1});
330 
331 
332 
333 //        // subtracting to find the relative distances
334 //        torch::Tensor distBetweenScatterersAndSensors =                   \
335 //            torch::linalg_norm(slottedCoordinates - slottedSensors,       \
336 //                               2, 2, true, torch::kFloat).to(DATATYPE);
337 
338 //        // substracting distance between relative fields
339 //        torch::Tensor distBetweenScatterersAndTransmitter =               \
340 //            torch::linalg_norm(slottedCoordinates - slotted_location,     \
341 //                               2, 2, true, torch::kFloat).to(DATATYPE);
342 
343 //        // adding up the distances
344 //        torch::Tensor distOfFlight   = \
345 //            distBetweenScatterersAndSensors + distBetweenScatterersAndTransmitter;
346 //        torch::Tensor timeOfFlight   = distOfFlight/1500;
347 //        torch::Tensor samplesOfFlight = \
348 //            torch::floor(timeOfFlight.squeeze() \
349 //            * this->sampling_frequency);
350 
351 
352 
353 //        // Adding pulses
354 //        #pragma omp parallel for
355 //        for(int sensor_index = 0; sensor_index < this->num_sensors; ++sensor_index){
356 //            for(int scatter_index = 0; scatter_index < samplesOfFlight[0].numel(); ++scatter_index){
357 
358 //                // getting the sample where the current scatter's contribution must be placed.
359 //                int where_to_place =                    \
```

```
360 //                  samplesOfFlight.index({sensor_index, \
361 //                                         scatter_index  \
362 //                                         }).item<int>();
363 
364 //              // checking whether that point is out of bounds
365 //              if(where_to_place >= numsamples) continue;
366 
367 //              // placing a reflectivity-scaled impulse in there
368 //              this->signalMatrix.index_put_({where_to_place, sensor_index},              \
369 //                                            this->signalMatrix.index({where_to_place,       \
370 //                                                                      sensor_index}) +      \
371 //                                              scatterers->reflectivity.index({0, \
372 //                                                                              scatter_index}));
373 //          }
374 //      }
375 
376 
377 
378 //      // // Adding pulses
379 //      // for(int sensor_index = 0; sensor_index < this->num_sensors; ++sensor_index){
380 
381 //      //    // indices associated with current index
382 //      //    torch::Tensor tensor_containing_placing_indices = \
383 //      //        samplesOfFlight[sensor_index].to(torch::kInt);
384 
385 //      //    // calculating histogram
386 //      //    auto uniqueOutputs = at::_unique(tensor_containing_placing_indices, false, true);
387 //      //    torch::Tensor bruh = std::get<1>(uniqueOutputs);
388 //      //    torch::Tensor uniqueValues = std::get<0>(uniqueOutputs).to(torch::kInt);
389 //      //    torch::Tensor uniqueCounts = torch::bincount(bruh).to(torch::kInt);
390 
391 //      //    // placing values according to histogram
392 //      //    this->signalMatrix.index_put_({uniqueValues.to(torch::kLong), sensor_index}, \
393 //      //                                  uniqueCounts.to(DATATYPE));
394 
395 //      // }
396 
397 //      // Creating matrix out of transmitted signal
398 //      torch::Tensor signalTensorAsArgument =         \
399 //          transmitterObj->Signal.reshape({
400 //              transmitterObj->Signal.numel(),        \
401 //              1});
402 //      signalTensorAsArgument =                       \
403 //          torch::tile(signalTensorAsArgument,        \
404 //                      {1, this->signalMatrix.size(1)});
405 
406 
407 
408 //      // convolving the pulse-matrix with the signal matrix
409 //      fConvolveColumns(this->signalMatrix,    \
410 //                       signalTensorAsArgument);
411 
412 
413 //      // trimming the convolved signal since the signal matrix length remains the same
414 //      this->signalMatrix = \
415 //          this->signalMatrix.index({
416 //              torch::indexing::Slice(0, numsamples), \
417 //              torch::indexing::Slice()});
418 
419 
420 //      // returning
421 //      return;
422 //  }
423 
424 //  /* ========================================================================
425 //  Aim: Decimating basebanded-received signal
426 //  ------------------------------------------------------------------------ */
427 //  void nfdc_decimateSignal(TransmitterClass* transmitterObj){
428 
429 //      // creating the matrix for frequency-shifting
430 //      torch::Tensor integerArray = torch::linspace(0, \
431 //                                                   this->signalMatrix.size(0)-1, \
432 //
    this->signalMatrix.size(0)).reshape({this->signalMatrix.size(0), 1});
433 //      integerArray             = torch::tile(integerArray, {1, this->num_sensors});
```

```
434 //        integerArray             = torch::exp(COMPLEX_1j * transmitterObj->fc * integerArray);
435
436 //        // storing original number of samples
437 //        int original_signalMatrix_numsamples = this->signalMatrix.size(0);
438
439 //        // producing frequency-shifting
440 //        this->signalMatrix       = torch::mul(this->signalMatrix, integerArray);
441
442 //        // low-pass filter
443 //        torch::Tensor lowpassfilter_impulseresponse =                    \
444 //            this->lowpassFilterCoefficientsForDecimation.reshape(        \
445 //                {this->lowpassFilterCoefficientsForDecimation.numel(),   \
446 //                1});
447 //        lowpassfilter_impulseresponse =                                  \
448 //            torch::tile(lowpassfilter_impulseresponse,                   \
449 //                        {1, this->signalMatrix.size(1)});
450
451 //        // low-pass filtering the signal
452 //        fConvolveColumns(this->signalMatrix,
453 //                        lowpassfilter_impulseresponse);
454
455 //        // Cutting down the extra-samples from convolution
456 //        this->signalMatrix = \
457 //            this->signalMatrix.index({torch::indexing::Slice(0, original_signalMatrix_numsamples), \
458 //                                      torch::indexing::Slice()});
459
460 //        // // Cutting off samples in the front.
461 //        // int cutoffpoint = lowpassfilter_impulseresponse.size(0) - 1;
462 //        // this->signalMatrix =                                          \
463 //        //     this->signalMatrix.index({                                \
464 //        //         torch::indexing::Slice(cutoffpoint,                   \
465 //        //                                torch::indexing::None),        \
466 //        //         torch::indexing::Slice()                              \
467 //        //     });
468
469 //        // building parameters for downsampling
470 //        int decimation_factor        = std::floor(this->sampling_frequency/transmitterObj->bandwidth);
471 //        this->decimation_factor      = decimation_factor;
472 //        this->post_decimation_sampling_frequency =                       \
473 //            this->sampling_frequency / this->decimation_factor;
474 //        int numsamples_after_decimation = std::floor(this->signalMatrix.size(0)/decimation_factor);
475
476 //        // building the samples which will be subsetted
477 //        torch::Tensor samplingIndices = \
478 //            torch::linspace(0, \
479 //                            numsamples_after_decimation * decimation_factor - 1, \
480 //                            numsamples_after_decimation).to(torch::kInt);
481
482 //        // downsampling the low-pass filtered signal
483 //        this->signalMatrix = \
484 //            this->signalMatrix.index({samplingIndices, \
485 //                                      torch::indexing::Slice()});
486
487 //        // returning
488 //        return;
489 //    }
490
491 //    /* ============================================================================
492 //    Aim: Match-filtering
493 //    ---------------------------------------------------------------------------- */
494 //    void nfdc_matchFilterDecimatedSignal(){
495
496 //        // Creating a 2D matrix out of the signal
497 //        torch::Tensor matchFilter2DMatrix = \
498 //            this->matchFilter.reshape({this->matchFilter.numel(), 1});
499 //        matchFilter2DMatrix = \
500 //            torch::tile(matchFilter2DMatrix, \
501 //                        {1, this->num_sensors});
502
503
504 //        // 2D convolving to produce the match-filtering
505 //        fConvolveColumns(this->signalMatrix, \
506 //                        matchFilter2DMatrix);
507
508
```

```
509 //        // Trimming the signal to contain just the signals that make sense to us
510 //        int startingpoint = matchFilter2DMatrix.size(0) - 1;
511 //        this->signalMatrix =                                \
512 //            this->signalMatrix.index({                      \
513 //                torch::indexing::Slice(startingpoint,       \
514 //                                torch::indexing::None), \
515 //                torch::indexing::Slice()});
516 
517 //        // // trimming the two ends of the signal
518 //        // int startingpoint = matchFilter2DMatrix.size(0) - 1;
519 //        // int endingpoint   = this->signalMatrix.size(0) \
520 //        //                     - matchFilter2DMatrix.size(0) \
521 //        //                     + 1;
522 //        // this->signalMatrix =                              \
523 //        //    this->signalMatrix.index({                     \
524 //        //        torch::indexing::Slice(startingpoint,      \
525 //        //                               endingpoint), \
526 //        //        torch::indexing::Slice()});
527 
528 
529 //    }
530 
531 //    /* ============================================================================
532 //    Aim: precompute delay-matrices
533 //    ---------------------------------------------------------------------------- */
534 //    void nfdc_precomputeDelayMatrices(TransmitterClass* transmitterObj){
535 
536 //        // calculating range-related parameters
537 //        int number_of_range_cells    = \
538 //            std::ceil(((this->recording_period * 1500)/2)/this->range_cell_size);
539 //        int number_of_azimuths_to_image = \
540 //            std::ceil(transmitterObj->azimuthal_beamwidth / this->azimuth_cell_size);
541 
542 //        // creating centers of range-cell centers
543 //        torch::Tensor range_centers = \
544 //            this->range_cell_size * \
545 //            torch::arange(0, number_of_range_cells) \
546 //            + this->range_cell_size/2;
547 //        this->range_centers = range_centers;
548 
549 //        // creating discretized azimuth-centers
550 //        torch::Tensor azimuth_centers =                \
551 //            this->azimuth_cell_size *                  \
552 //            torch::arange(0, number_of_azimuths_to_image) \
553 //            + this->azimuth_cell_size/2;
554 //        this->azimuth_centers = azimuth_centers;
555 //        this->azimuth_centers = this->azimuth_centers - torch::mean(this->azimuth_centers);
556 
557 //        // finding the mesh values
558 //        auto range_azimuth_meshgrid = \
559 //            torch::meshgrid({range_centers, \
560 //                             azimuth_centers}, "ij");
561 //        torch::Tensor range_grid = range_azimuth_meshgrid[0]; // the columns are range_centers
562 //        torch::Tensor azimuth_grid = range_azimuth_meshgrid[1]; // the rows are azimuth-centers
563 
564 //        // going from 2D to 3D
565 //        range_grid = \
566 //            torch::tile(range_grid.reshape({range_grid.size(0), \
567 //                                            range_grid.size(1), \
568 //                                            1}), {1,1,this->num_sensors});
569 //        azimuth_grid = \
570 //            torch::tile(azimuth_grid.reshape({azimuth_grid.size(0), \
571 //                                              azimuth_grid.size(1), \
572 //                                              1}), {1, 1, this->num_sensors});
573 
574 //        // creating x_m tensor
575 //        torch::Tensor sensorCoordinatesSlot = \
576 //            this->inter_element_spacing * \
577 //            torch::arange(0, this->num_sensors).reshape({
578 //                1, 1, this->num_sensors
579 //            }).to(DATATYPE);
580 
581 //        sensorCoordinatesSlot = \
582 //            torch::tile(sensorCoordinatesSlot, \
583 //                        {range_grid.size(0),\
```

```
584 //                      range_grid.size(1),
585 //                      1});
586 
587 //      // calculating distances
588 //      torch::Tensor distanceMatrix =                              \
589 //          torch::square(range_grid - sensorCoordinatesSlot) +     \
590 //          torch::mul((2 * sensorCoordinatesSlot),                 \
591 //                    torch::mul(range_grid,                        \
592 //                             1 - torch::cos(azimuth_grid * PI/180)));
593 //      distanceMatrix = torch::sqrt(distanceMatrix);
594 
595 //      // finding the time taken
596 //      torch::Tensor timeMatrix = distanceMatrix/1500;
597 //      torch::Tensor sampleMatrix = timeMatrix * this->sampling_frequency;
598 
599 //      // finding the delay to be given
600 //      auto bruh390             = torch::max(sampleMatrix, 2, true);
601 //      torch::Tensor max_delay  = std::get<0>(bruh390);
602 //      torch::Tensor delayMatrix = max_delay - sampleMatrix;
603 
604 //      // now that we have the delay entries, we need to create the matrix that does it
605 //      int decimation_factor = \
606 //          std::floor(static_cast<float>(this->sampling_frequency)/transmitterObj->bandwidth);
607 //      this->decimation_factor = decimation_factor;
608 
609 
610 //      // calculating frame-size
611 //      int frame_size = \
612 //          std::ceil(static_cast<float>((2 * this->range_cell_size / 1500) * \
613 //                    static_cast<float>(this->sampling_frequency)/decimation_factor));
614 //      this->frame_size = frame_size;
615 
616 //      // // calculating the buffer-zeros to add
617 //      // int num_buffer_zeros_per_frame = \
618 //      //     static_cast<float>(this->num_sensors - 1) * \
619 //      //     static_cast<float>(this->inter_element_spacing) * \
620 //      //     this->sampling_frequency /1500;
621 
622 //      int num_buffer_zeros_per_frame =               \
623 //          std::ceil((this->num_sensors - 1) *        \
624 //                    this->inter_element_spacing *    \
625 //                    this->sampling_frequency         \
626 //                    / (1500 * this->decimation_factor));
627 
628 //      // storing to class member
629 //      this->num_buffer_zeros_per_frame = \
630 //          num_buffer_zeros_per_frame;
631 
632 //      // calculating the total frame-size
633 //      int total_frame_size = \
634 //          this->frame_size + this->num_buffer_zeros_per_frame;
635 
636 //      // creating the multiplication matrix
637 //      torch::Tensor mulFFTMatrix = \
638 //          torch::linspace(0, \
639 //                          total_frame_size-1, \
640 //                          total_frame_size).reshape({1, \
641 //                                                     total_frame_size, \
642 //                                                     1}).to(DATATYPE); // creating an array 1,...,frame_size
    of shape [1,frame_size, 1];
643 //      mulFFTMatrix = \
644 //          torch::div(mulFFTMatrix, \
645 //                    torch::tensor(total_frame_size).to(DATATYPE)); // dividing by N
646 //      mulFFTMatrix = mulFFTMatrix * 2 * PI * -1 * COMPLEX_1j; // creating tenosr values for -1j * 2pi *
    k/N
647 //      mulFFTMatrix = \
648 //          torch::tile(mulFFTMatrix, \
649 //                     {number_of_range_cells * number_of_azimuths_to_image, \
650 //                      1, \
651 //                      this->num_sensors}); // creating the larger tensor for it
652 
653 
654 //      // populating the matrix
655 //      for(int azimuth_index = 0; \
656 //          azimuth_index<number_of_azimuths_to_image; \
```

```
657 //            ++azimuth_index){
658 //            for(int range_index = 0; \
659 //                range_index < number_of_range_cells; \
660 //                ++range_index){
661 //                // finding the delays for sensors
662 //                torch::Tensor currentSensorDelays = \
663 //                    delayMatrix.index({range_index, \
664 //                                       azimuth_index, \
665 //                                       torch::indexing::Slice()});
666 //                // reshaping it to the target size
667 //                currentSensorDelays = \
668 //                    currentSensorDelays.reshape({1, \
669 //                                                 1, \
670 //                                                 this->num_sensors});
671 //                // tiling across the plane
672 //                currentSensorDelays = \
673 //                    torch::tile(currentSensorDelays, \
674 //                                {1, total_frame_size, 1});
675 //                // multiplying across the appropriate plane
676 //                int index_to_place_at = \
677 //                    azimuth_index * number_of_range_cells + \
678 //                    range_index;
679 //                mulFFTMatrix.index_put_({index_to_place_at, \
680 //                                         torch::indexing::Slice(),
681 //                                         torch::indexing::Slice()}, \
682 //                                         currentSensorDelays);
683 //            }
684 //        }
685 
686 //        // storing the mulFFTMatrix
687 //        this->mulFFTMatrix = mulFFTMatrix;
688 //    }
689 
690 //    /* ============================================================================
691 //    Aim: Beamforming the signal
692 //    ---------------------------------------------------------------------------- */
693 //    void nfdc_beamforming(TransmitterClass* transmitterObj){
694 
695 //        // ensuring the signal matrix is in the shape we want
696 //        if(this->signalMatrix.size(1) != this->num_sensors)
697 //            throw std::runtime_error("The second dimension doesn't correspond to the number of sensors \n");
698 
699 //        // adding the batch-dimension
700 //        this->signalMatrix = \
701 //            this->signalMatrix.reshape({               \
702 //                1,                                     \
703 //                this->signalMatrix.size(0),            \
704 //                this->signalMatrix.size(1)});
705 
706 
707 //        // zero-padding to ensure correctness
708 //        int ideal_length = \
709 //            std::ceil(this->range_centers.numel() * this->frame_size);
710 //        int num_zeros_to_pad_signal_along_dimension_0 = \
711 //            ideal_length - this->signalMatrix.size(1);
712 
713 
714 //        // printing
715 //        if (DEBUG_ULA) PRINTSMALLLINE
716 //        if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->range_centers.numel()     =
    "<<this->range_centers.numel() <<std::endl;
717 //        if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->frame_size               =
    "<<this->frame_size             <<std::endl;
718 //        if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | ideal_length                  =
    "<<ideal_length                 <<std::endl;
719 //        if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->signalMatrix.size(1)     =
    "<<this->signalMatrix.size(1)   <<std::endl;
720 //        if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming |
    num_zeros_to_pad_signal_along_dimension_0 = "<<num_zeros_to_pad_signal_along_dimension_0 <<std::endl;
721 //        if (DEBUG_ULA) PRINTSPACE
722 
723 //        // appending or slicing based on the requirements
724 //        if (num_zeros_to_pad_signal_along_dimension_0 <= 0) {
725 
726 //            // sending out a warning that slicing is going on
```

```
727 //          if (DEBUG_ULA) std::cerr <<"\t\t ULAClass::nfdc_beamforming | Please note that the signal
    matrix has been sliced. This could lead to loss of information"<<std::endl;
728
729 //          // slicing the signal matrix
730 //          if (DEBUG_ULA) PRINTSPACE
731 //          if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->signalMatrix.shape (before
    slicing) = "<< this->signalMatrix.sizes().vec() <<std::endl;
732 //          this->signalMatrix = \
733 //              this->signalMatrix.index({torch::indexing::Slice(), \
734 //                                        torch::indexing::Slice(0, ideal_length), \
735 //                                        torch::indexing::Slice()});
736 //          if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->signalMatrix.shape (after
    slicing) = "<< this->signalMatrix.sizes().vec() <<std::endl;
737 //          if (DEBUG_ULA) PRINTSPACE
738
739 //      }
740 //      else {
741 //          // creating a zero-filled tensor to append to signal matrix
742 //          torch::Tensor zero_tensor =                                 \
743 //              torch::zeros({this->signalMatrix.size(0),               \
744 //                            num_zeros_to_pad_signal_along_dimension_0, \
745 //                            this->num_sensors}).to(DATATYPE);
746
747 //          // appending to signal matrix
748 //          this->signalMatrix       =                                  \
749 //              torch::cat({this->signalMatrix, zero_tensor}, 1);
750 //      }
751
752 //      // breaking the signal into frames
753 //      fBuffer2D(this->signalMatrix, frame_size);
754
755
756 //      // add some zeros to the end of frames to accomodate delaying of signals.
757 //      torch::Tensor zero_filled_tensor =                  \
758 //          torch::zeros({this->signalMatrix.size(0),       \
759 //                        this->num_buffer_zeros_per_frame, \
760 //                        this->num_sensors}).to(DATATYPE);
761 //      this->signalMatrix =                                \
762 //          torch::cat({this->signalMatrix,                 \
763 //                      zero_filled_tensor}, 1);
764
765
766 //      // tiling it to ensure that it works for all range-angle combinations
767 //      int number_of_azimuths_to_image = this->azimuth_centers.numel();
768 //      this->signalMatrix = \
769 //          torch::tile(this->signalMatrix, \
770 //                      {number_of_azimuths_to_image, 1, 1});
771
772 //      // element-wise multiplying the signals to delay each of the frame accordingly
773 //      this->signalMatrix = torch::mul(this->signalMatrix, \
774 //                                      this->mulFFTMatrix);
775
776 //      // summing up the signals
777 //      // this->signalMatrix = torch::sum(this->signalMatrix, \
778 //      //                                 2,                  \
779 //      //                                 true);
780 //      this->signalMatrix = torch::sum(this->signalMatrix, \
781 //                                      2,                  \
782 //                                      false);
783
784
785 //      // printing some stuff
786 //      if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming: this->azimuth_centers.numel() =
    "<<this->azimuth_centers.numel() <<std::endl;
787 //      if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming: this->range_centers.numel() =
    "<<this->range_centers.numel() <<std::endl;
788 //      if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming: total number              =
    "<<this->range_centers.numel() * this->azimuth_centers.numel() <<std::endl;
789 //      if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming: this->signalMatrix.sizes().vec() =
    "<<this->signalMatrix.sizes().vec() <<std::endl;
790
791 //      // creating a tensor to store the final image
792 //      torch::Tensor finalImage = \
793 //          torch::zeros({this->frame_size * this->range_centers.numel(), \
794 //                        this->azimuth_centers.numel()}).to(torch::kComplexFloat);
```

```
795
796
797 //        // creating a loop to assign values
798 //        for(int range_index = 0; range_index < this->range_centers.numel(); ++range_index){
799 //            for(int angle_index = 0; angle_index < this->azimuth_centers.numel(); ++angle_index){
800
801 //                // getting row index
802 //                int rowindex = \
803 //                    angle_index * this->range_centers.numel() \
804 //                    + range_index;
805
806 //                // getting the strip to store
807 //                torch::Tensor strip = \
808 //                    this->signalMatrix.index({rowindex, \
809 //                                              torch::indexing::Slice()});
810
811 //                // taking just the first few values
812 //                strip = strip.index({torch::indexing::Slice(0, this->frame_size)});
813
814 //                // placing the strips on the image
815 //                finalImage.index_put_({\
816 //                    torch::indexing::Slice((range_index)*this->frame_size, \
817 //                                           (range_index+1)*this->frame_size), \
818 //                                          angle_index}, \
819 //                                          strip);
820
821 //            }
822 //        }
823
824 //        // saving the image
825 //        this->beamformedImage = finalImage;
826
827
828
829 //        // converting image from polar to cartesian
830 //        nfdc_PolarToCartesian();
831
832
833 //    }
834
835 //    /* ========================================================================
836 //    Aim: Converting Polar Image to Cartesian
837 //    ........................................................................
838 //    Note:
839 //        > For now, we're assuming that the r value is one.
840 //    ------------------------------------------------------------------------ */
841 //    void nfdc_PolarToCartesian(){
842
843
844 //        // deciding image dimensions
845 //        int num_pixels_width  = 128;
846 //        int num_pixels_height = 128;
847
848
849 //        // creating query points
850 //        torch::Tensor max_right =                                       \
851 //            torch::cos(                                                 \
852 //                torch::max(                                             \
853 //                    this->azimuth_centers                               \
854 //                    - torch::mean(this->azimuth_centers)                \
855 //                    + torch::tensor({90}).to(DATATYPE))             \
856 //                * PI/180);
857 //        torch::Tensor max_left  =                                       \
858 //            torch::cos(                                                 \
859 //                torch::min(this->azimuth_centers                        \
860 //                          - torch::mean(this->azimuth_centers)          \
861 //                          + torch::tensor({90}).to(DATATYPE))       \
862 //                * PI/180);
863 //        torch::Tensor max_top   = torch::tensor({1});
864 //        torch::Tensor max_bottom = torch::min(this->range_centers);
865
866
867 //        // creating query points along the x-dimension
868 //        torch::Tensor query_x =                         \
869 //            torch::linspace(                            \
```

```
870 //                max_left,                               \
871 //                max_right,                              \
872 //                num_pixels_width                        \
873 //                ).to(DATATYPE);
874 
875 //        torch::Tensor query_y =                         \
876 //            torch::linspace(                            \
877 //                max_bottom.item<float>(),               \
878 //                max_top.item<float>(),                  \
879 //                num_pixels_height                       \
880 //                ).to(DATATYPE);
881 
882 
883 //        // converting original coordinates to their corresponding cartesian
884 //        float delta_r = 1/static_cast<float>(this->beamformedImage.size(0));
885 //        float delta_azimuth =                           \
886 //            torch::abs(                                 \
887 //                this->azimuth_centers.index({1})        \
888 //                - this->azimuth_centers.index({0})      \
889 //                ).item<float>();
890 
891 
892 
893 //        // getting query points
894 //        torch::Tensor range_values = \
895 //            torch::linspace(                            \
896 //                delta_r,                                \
897 //                this->beamformedImage.size(0) * delta_r, \
898 //                this->beamformedImage.size(0)           \
899 //                ).to(DATATYPE);
900 //        range_values = \
901 //            range_values.reshape({range_values.numel(), 1});
902 //        range_values = \
903 //            torch::tile(range_values, \
904 //                        {1, this->azimuth_centers.numel()});
905 
906 
907 //        // getting angle-values
908 //        torch::Tensor angle_values =                    \
909 //            this->azimuth_centers                       \
910 //            - torch::mean(this->azimuth_centers)        \
911 //            + torch::tensor({90});
912 //        angle_values =                                  \
913 //            torch::tile(                                \
914 //                angle_values,                           \
915 //                {this->beamformedImage.size(0), 1});
916 
917 
918 //        // converting to cartesian original points
919 //        torch::Tensor query_original_x = \
920 //            range_values * torch::cos(angle_values * PI/180);
921 //        torch::Tensor query_original_y = \
922 //            range_values * torch::sin(angle_values * PI/180);
923 
924 
925 
926 //        // converting points to vector 2D format
927 //        torch::Tensor query_source =                                        \
928 //            torch::cat({                                                    \
929 //                query_original_x.reshape({1, query_original_x.numel()}),  \
930 //                query_original_y.reshape({1, query_original_y.numel()})}, \
931 //                0);
932 
933 
934 //        // converting reflectivity to corresponding 2D format
935 //        torch::Tensor reflectivity_vectors = \
936 //            this->beamformedImage.reshape({1, this->beamformedImage.numel()});
937 
938 
939 //        // creating image
940 //        torch::Tensor cartesianImageLocal =             \
941 //            torch::zeros(                               \
942 //                {num_pixels_height,                     \
943 //                 num_pixels_width}                      \
944 //                 ).to(torch::kComplexFloat);
```

```
945
946 //        /*
947 //        Next Aim: start interpolating the points on the uniform grid.
948 //        */
949 //        #pragma omp parallel for
950 //        for(int x_index = 0; x_index < query_x.numel(); ++x_index){
951 //            // if(DEBUG_ULA) std::cout << "\t\t\t x_index = " << x_index << " ";
952 //            #pragma omp parallel for
953 //            for(int y_index = 0; y_index < query_y.numel(); ++y_index){
954 //                // if(DEBUG_ULA) if(y_index%16 == 0) std::cout<<".";
955
956 //                // getting current values
957 //                torch::Tensor current_x = query_x.index({x_index}).reshape({1, 1});
958 //                torch::Tensor current_y = query_y.index({y_index}).reshape({1, 1});
959
960
961
962
963 //                // getting the query value
964 //                torch::Tensor query_vector = torch::cat({current_x, current_y}, 0);
965
966
967 //                // copying the query source
968 //                torch::Tensor query_source_relative = query_source;
969 //                query_source_relative = query_source_relative - query_vector;
970
971
972 //                // subsetting using absolute values and masks
973 //                float threshold = delta_r * 10;
974 //                // PRINTDOTS
975 //                auto mask_row = \
976 //                    torch::abs(query_source_relative[0]) <= threshold;
977 //                auto mask_col = \
978 //                    torch::abs(query_source_relative[1]) <= threshold;
979 //                auto mask_together = torch::mul(mask_row, mask_col);
980
981
982
983
984 //                // calculating number of points in threshold neighbourhood
985 //                int num_points_in_threshold_neighbourhood = \
986 //                    torch::sum(mask_together).item<int>();
987 //                if (num_points_in_threshold_neighbourhood == 0){
988 //                    continue;
989 //                }
990
991
992
993 //                // subsetting points in neighbourhood
994 //                torch::Tensor PointsInNeighbourhood =   \
995 //                    query_source_relative.index({
996 //                        torch::indexing::Slice(),       \
997 //                        mask_together});
998 //                torch::Tensor ReflectivitiesInNeighbourhood = \
999 //                    reflectivity_vectors.index({torch::indexing::Slice(), mask_together});
1000
1001
1002 //                // finding the distance between the points
1003 //                torch::Tensor relativeDistances = \
1004 //                    torch::linalg_norm(PointsInNeighbourhood, \
1005 //                                       2, 0, true, \
1006 //                                       torch::kFloat).to(DATATYPE);
1007
1008
1009 //                // calculating weighing factor
1010 //                torch::Tensor weighingFactor =                          \
1011 //                    torch::nn::functional::softmax(                     \
1012 //                        torch::max(relativeDistances)- relativeDistances, \
1013 //                        torch::nn::functional::SoftmaxFuncOptions(1));
1014
1015
1016 //                // combining intensities using distances
1017 //                torch::Tensor finalIntensity = \
1018 //                    torch::sum(
1019 //                        torch::mul(weighingFactor, \
```

```
1020 //                                ReflectivitiesInNeighbourhood));
1021
1022 //                // assigning values
1023 //                cartesianImageLocal.index_put_({x_index, y_index}, finalIntensity);
1024
1025 //            }
1026 //            // std::cout<<std::endl;
1027 //        }
1028
1029 //        // saving to member function
1030 //        this->cartesianImage = cartesianImageLocal;
1031
1032 //    }
1033
1034 //    /* ============================================================================
1035 //    Aim: create acoustic image directly
1036 //    ---------------------------------------------------------------------------- */
1037 //    void nfdc_createAcousticImage(ScattererClass* scatterers, \
1038 //                                TransmitterClass* transmitterObj){
1039
1040 //        // first we ensure that the scattersers are in our frame of reference
1041 //        scatterers->coordinates = scatterers->coordinates - this->location;
1042
1043 //        // finding the spherical coordinates of the scatterers
1044 //        torch::Tensor scatterers_spherical = fCart2Sph(scatterers->coordinates);
1045
1046 //        // note that its not precisely projection. its rotation. So the original lengths must be
     maintained. but thats easy since the operation of putting th eelevation to be zero works just fine.
1047 //        scatterers_spherical.index_put_({1, torch::indexing::Slice()}, 0);
1048
1049 //        // converting the points back to cartesian
1050 //        torch::Tensor scatterers_acoustic_cartesian = fSph2Cart(scatterers_spherical);
1051
1052 //        // removing the z-dimension
1053 //        scatterers_acoustic_cartesian = \
1054 //            scatterers_acoustic_cartesian.index({torch::indexing::Slice(0, 2, 1), \
1055 //                                        torch::indexing::Slice()});
1056
1057 //        // deciding image dimensions
1058 //        int num_pixels_x = 512;
1059 //        int num_pixels_y = 512;
1060 //        torch::Tensor acousticImage =                   \
1061 //            torch::zeros({num_pixels_x,                 \
1062 //                        num_pixels_y}).to(DATATYPE);
1063
1064 //        // finding the max and min values
1065 //        torch::Tensor min_x   = torch::min(scatterers_acoustic_cartesian[0]);
1066 //        torch::Tensor max_x   = torch::max(scatterers_acoustic_cartesian[0]);
1067 //        torch::Tensor min_y   = torch::min(scatterers_acoustic_cartesian[1]);
1068 //        torch::Tensor max_y   = torch::max(scatterers_acoustic_cartesian[1]);
1069
1070 //        // creating query grids
1071 //        torch::Tensor query_x = torch::linspace(0, 1, num_pixels_x);
1072 //        torch::Tensor query_y = torch::linspace(0, 1, num_pixels_y);
1073
1074 //        // scaling it up to image max-point spread
1075 //        query_x           = min_x + (max_x - min_x) * query_x;
1076 //        query_y           = min_y + (max_y - min_y) * query_y;
1077 //        float delta_queryx = (query_x[1] - query_x[0]).item<float>();
1078 //        float delta_queryy = (query_y[1] - query_y[0]).item<float>();
1079
1080 //        // creating a mesh-grid
1081 //        auto queryMeshGrid = torch::meshgrid({query_x, query_y}, "ij");
1082 //        query_x = queryMeshGrid[0].reshape({1, queryMeshGrid[0].numel()});
1083 //        query_y = queryMeshGrid[1].reshape({1, queryMeshGrid[1].numel()});;
1084 //        torch::Tensor queryMatrix = torch::cat({query_x, query_y}, 0);
1085
1086 //        // printing shapes
1087 //        if(DEBUG_ULA) PRINTSMALLLINE
1088 //        if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: query_x.shape =
     "<<query_x.sizes().vec()<<std::endl;
1089 //        if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: query_y.shape =
     "<<query_y.sizes().vec()<<std::endl;
1090 //        if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: queryMatrix.shape =
     "<<queryMatrix.sizes().vec()<<std::endl;
```

```
1091
1092 //        // setting up threshold values
1093 //        float threshold_value =      \
1094 //            std::min(delta_queryx,   \
1095 //                     delta_queryy); if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage:
     line 711"<<std::endl;
1096
1097 //        // putting a loop through the whole thing
1098 //        for(int i = 0; i<queryMatrix[0].numel(); ++i){
1099 //            // for each element in the query matrix
1100 //            if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 716"<<std::endl;
1101
1102 //            // calculating relative position of all the points
1103 //            torch::Tensor relativeCoordinates = \
1104 //                scatterers_acoustic_cartesian - \
1105 //                queryMatrix.index({torch::indexing::Slice(), i}).reshape({2, 1}); if(DEBUG_ULA)
     std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 720"<<std::endl;
1106
1107 //            // calculating distances between all the points and the query point
1108 //            torch::Tensor relativeDistances = \
1109 //                torch::linalg_norm(relativeCoordinates, \
1110 //                                   1, 0, true, \
1111 //                                   DATATYPE);if(DEBUG_ULA) std::cout<<"\t\t\t
     ULAClass::nfdc_createAcousticImage: line 727"<<std::endl;
1112 //            // finding points that are within the threshold
1113 //            torch::Tensor conditionMeetingPoints = \
1114 //                relativeDistances.squeeze() <= threshold_value;if(DEBUG_ULA) std::cout<<"\t\t\t
     ULAClass::nfdc_createAcousticImage: line 729"<<std::endl;
1115
1116 //            // subsetting the points in the neighbourhood
1117 //            if(torch::sum(conditionMeetingPoints).item<float>() == 0){
1118
1119 //                // continuing implementation if there are no points in the neighbourhood
1120 //                continue; if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line
     735"<<std::endl;
1121 //            }
1122 //            else{
1123 //                // creating mask for points in the neighbourhood
1124 //                auto mask = (conditionMeetingPoints == 1);if(DEBUG_ULA) std::cout<<"\t\t\t
     ULAClass::nfdc_createAcousticImage: line 739"<<std::endl;
1125
1126 //                // subsetting relative distances in the neighbourhood
1127 //                torch::Tensor distanceInTheNeighbourhood = \
1128 //                    relativeDistances.index({torch::indexing::Slice(), mask});if(DEBUG_ULA)
     std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 743"<<std::endl;
1129
1130 //                // subsetting reflectivity of points in the neighbourhood
1131 //                torch::Tensor reflectivityInTheNeighbourhood = \
1132 //                    scatterers->reflectivity.index({torch::indexing::Slice(), mask});if(DEBUG_ULA)
     std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 747"<<std::endl;
1133
1134 //                // assigning intensity as a function of distance and reflectivity
1135 //                torch::Tensor reflectivityAssignment =                    \
1136 //                    torch::mul(torch::exp(-distanceInTheNeighbourhood),   \
1137 //                               reflectivityInTheNeighbourhood);if(DEBUG_ULA) std::cout<<"\t\t\t
     ULAClass::nfdc_createAcousticImage: line 752"<<std::endl;
1138 //                reflectivityAssignment = \
1139 //                    torch::sum(reflectivityAssignment);if(DEBUG_ULA) std::cout<<"\t\t\t
     ULAClass::nfdc_createAcousticImage: line 754"<<std::endl;
1140
1141 //                // assigning this value to the image pixel intensity
1142 //                int pixel_position_x = i%num_pixels_x;
1143 //                int pixel_position_y = std::floor(i/num_pixels_x);
1144 //                acousticImage.index_put_({pixel_position_x, \
1145 //                                          pixel_position_y}, \
1146 //                                         reflectivityAssignment.item<float>());if(DEBUG_ULA)
     std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 761"<<std::endl;
1147 //            }
1148
1149 //        }
1150
1151 //        // storing the acoustic-image to the member
1152 //        this->currentArtificalAcousticImage = acousticImage;
1153
1154 //        // // saving the torch::tensor
```

```
//        // torch::save(acousticImage, \
//        //            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/Assets/acoustic_image.pt");


//        // // bringing it back to the original coordinates
//        // scatterers->coordinates = scatterers->coordinates + this->location;
//    }



// };


template <typename T>
class ULAClass
{
public:
    // intrinsic parameters
    int                         num_sensors;                        // number of sensors
    T                           inter_element_spacing;              // space between sensors
    std::vector<std::vector<T>> coordinates;                        // coordinates of each sensor
    T                           sampling_frequency;                 // sampling frequency of the sensors
    T                           recording_period;                   // recording period of the ULA
    std::vector<T>              location;                           // location of first coordinate

    // derived
    std::vector<T>              sensor_direction;
    std::vector<std::vector<T>> signalMatrix;

    // decimation related
    int              decimation_factor;                         // the new decimation factor
    T                post_decimation_sampling_frequency;        // the new sampling frequency
    std::vector<T>   lowpass_filter_coefficients_for_decimation; // filter-coefficients for filtering

    // imaging related
    T  range_resolution;           // theoretical range-resolution = $\frac{c}{2B}$
    T  azimuthal_resolution;       // theoretical azimuth-resolution =
        $\frac{\lambda}{(N-1)*inter-element-distance}$
    T  range_cell_size;            // the range-cell quanta we're choosing for efficiency trade-off
    T  azimuth_cell_size;          // the azimuth quanta we're choosing
    std::vector<T>  azimuth_centers; // tensor containing the azimuth centeres
    std::vector<T>  range_centers;  // tensor containing the range-centers
    int frame_size;                // the frame-size corresponding to a range cell in a decimated signal
        matrix

    std::vector<std::vector<complex<T>>> mulFFTMatrix; // the matrix containing the delays for each-element
        as a slot
    std::vector<complex<T>>            matchFilter;  // torch tensor containing the match-filter
    int   num_buffer_zeros_per_frame;                // number of zeros we're adding per frame to ensure
        no-rotation
    std::vector<std::vector<T>> beamformedImage;     // the beamformed image
    std::vector<std::vector<T>> cartesianImage;      // the cartesian version of beamformed image

    // Artificial acoustic-image related
    std::vector<std::vector<T>> currentArtificialAcousticImage; // acoustic image directly produced


    // Basic Constructor
    ULAClass() = default;

    // constructor
    ULAClass(const int    num_sensors_arg,
             const auto   inter_element_spacing_arg,
             const auto&  coordinates_arg,
             const auto&  sampling_frequency_arg,
             const auto&  recording_period_arg,
             const auto&  location_arg,
             const auto&  signalMatrix_arg,
             const auto&  lowpass_filter_coefficients_for_decimation_arg):
               num_sensors(num_sensors_arg),
               inter_element_spacing(inter_element_spacing_arg),
               coordinates(std::move(coordinates_arg)),
               sampling_frequency(sampling_frequency_arg),
               recording_period(recording_period_arg),
```

```
1226                location(std::move(location_arg)),
1227                signalMatrix(std::move(signalMatrix_arg)),
1228                lowpass_filter_coefficients_for_decimation(std::move(lowpass_filter_coefficients_for_decimation_arg))
1229        {
1230
1231            // calculating ULA direction
1232            sensor_direction = std::vector<T>{coordinates[1][0] - coordinates[0][0],
1233                                              coordinates[1][1] - coordinates[0][1],
1234                                              coordinates[1][2] - coordinates[0][2]};
1235
1236            // normalizing
1237            auto    norm_value_temp   {std::inner_product(sensor_direction.begin(), sensor_direction.end(),
1238                                                          sensor_direction.begin(),
1239                                                          0.00)};
1240
1241            // dividing
1242            if (norm_value_temp != 0) {sensor_direction =  sensor_direction / norm_value_temp;}
1243
1244        }
1245
1246        // deleting copy constructor/assignment
1247        // ULAClass(const ULAClass& other)                  = delete;
1248        // ULAClass&  operator=(const   ULAClass&  other)     = delete;
1249
1250
1251    };
```

8.1.4 Class: Autonomous Underwater Vehicle

The following is the class definition used to encapsulate attributes and methods of the marine vessel.

```
1
2
3   #pragma once
4
5   // function to plot the thing
6   void fPlotTensors(){
7       system("python /Users/vrsreeganesh/Documents/GitHub/AUV/Code/Python/TestingSaved_tensors.py");
8   }
9
10
11  void fSaveSeafloorScatteres(ScattererClass scatterer, \
12                              ScattererClass scatterer_fls, \
13                              ScattererClass scatterer_port, \
14                              ScattererClass scatterer_starboard){
15
16      // saving the ground-truth
17      ScattererClass SeafloorScatter_gt = scatterer;
18      save(SeafloorScatter_gt.coordinates,
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt.pt");
19      save(SeafloorScatter_gt.reflectivity,
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt_reflectivity.pt");
20
21      // saving coordinates
22      save(scatterer_fls.coordinates,
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_fls_coordinates.pt");
23      save(scatterer_port.coordinates,
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_port_coordinates.pt");
24      save(scatterer_starboard.coordinates,
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_starboard.coordinates.pt");
25
26      // saving reflectivities
27      save(scatterer_fls.reflectivity,
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_fls_coordinates_reflectivity.pt");
28      save(scatterer_port.reflectivity,
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_port_coordinates_reflectivity.pt");
29      save(scatterer_starboard.reflectivity,
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_starboard.coordinates_reflectivity.pt");
30
31      // plotting tensors
32      fPlotTensors();
33
34      // // saving the tensors
35      // if (true) {
36
37      //     // getting time ID
38      //     auto timeID = fGetCurrentTimeFormatted();
39
40      //     cout<<"\t\t\t\t\t\t\t Saving Tensors (timeID: "<<timeID<<")"<<endl;
41
42      //     // saving the ground-truth
43      //     ScattererClass SeafloorScatter_gt = scatterer;
44      //     save(SeafloorScatter_gt.coordinates, \
45      //                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt.pt");
46      //     save(SeafloorScatter_gt.reflectivity, \
47      //
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt_reflectivity.pt");
48
49
50      //     // saving coordinates
51      //     save(scatterer_fls.coordinates, \
52      //
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_fls_coordinates.pt");
53      //     save(scatterer_port.coordinates, \
54      //
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_port_coordinates.pt");
55      //     save(scatterer_starboard.coordinates, \
56      //
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_starboard.coordinates.pt");
```

```
57
58      //    // saving reflectivities
59      //    save(scatterer_fls.reflectivity, \
60      //
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_fls_coordinates_reflectivity.pt");
61      //    save(scatterer_port.reflectivity, \
62      //
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_port_coordinates_reflectivity.pt");
63      //    save(scatterer_starboard.reflectivity, \
64      //
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_starboard.coordinates_reflectivity.pt");
65
66      //    // plotting tensors
67      //    fPlotTensors();
68
69      //    // indicating end of thread
70      //    cout<<"\t\t\t\t\t\t\t Ended (timeID: "<<timeID<<")"<<endl;
71      // }
72  }
73
74
75  // hash-defines
76  #define PI                              3.14159265
77  #define DEBUGMODE_AUV                   false
78  #define SAVE_SIGNAL_MATRIX              true
79  #define SAVE_DECIMATED_SIGNAL_MATRIX    true
80  #define SAVE_MATCHFILTERED_SIGNAL_MATRIX true
81
82  class AUVClass{
83  public:
84      // Intrinsic attributes
85      Tensor location;                // location of vessel
86      Tensor velocity;                // current speed of the vessel [a vector]
87      Tensor acceleration;            // current acceleration of vessel [a vector]
88      Tensor pointing_direction;      // direction to which the AUV is pointed
89
90      // uniform linear-arrays
91      ULAClass ULA_fls;                       // front-looking SONAR ULA
92      ULAClass ULA_port;                      // mounted ULA [object of class, ULAClass]
93      ULAClass ULA_starboard;                 // mounted ULA [object of class, ULAClass]
94
95      // transmitters
96      TransmitterClass transmitter_fls;       // transmitter for front-looking SONAR
97      TransmitterClass transmitter_port;      // mounted transmitter [obj of class, TransmitterClass]
98      TransmitterClass transmitter_starboard; // mounted transmitter [obj of class, TransmitterClass]
99
100     // derived or dependent attributes
101     Tensor signalMatrix_1;          // matrix containing the signals obtained from ULA_1
102     Tensor largeSignalMatrix_1;     // matrix holding signal of synthetic aperture
103     Tensor beamformedLargeSignalMatrix; // each column is the beamformed signal at each stop-hop
104
105     // plotting mode
106     bool plottingmode;  // to suppress plotting associated with classes
107
108     // spotlight mode related
109     Tensor absolute_coords_patch_cart; // cartesian coordinates of patch
110
111     // Synthetic Aperture Related
112     Tensor ApertureSensorLocations; // sensor locations of aperture
113
114
115
116
117
118
119
120
121     /* ============================================================================
122     Aim: Init
123     ----------------------------------------------------------------------------*/
124     void init(){
125
126         // call sync-component attributes
127         this->syncComponentAttributes();
128         if (DEBUGMODE_AUV) cout << "AUVCLass::init: line 128" << endl;
```

```
129
130         // initializing all the ULAs
131         this->ULA_fls.init(      &this->transmitter_fls);
132         this->ULA_port.init(     &this->transmitter_port);
133         this->ULA_starboard.init( &this->transmitter_starboard);
134         if (DEBUGMODE_AUV) cout << "AUVCLass::init: line 134" << endl;
135
136
137         // precomputing delay-matrices for the ULA-class
138         thread ULA_fls_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
139                                             &this->ULA_fls,                          \
140                                             &this->transmitter_fls);
141         thread ULA_port_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
142                                              &this->ULA_port,                         \
143                                              &this->transmitter_port);
144         thread ULA_starboard_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
145                                                   &this->ULA_starboard,                \
146                                                   &this->transmitter_starboard);
147         if (DEBUGMODE_AUV) cout << "AUVCLass::init: line 145" << endl;
148
149         // joining the threads back
150         ULA_fls_precompute_weights_t.join();
151         ULA_port_precompute_weights_t.join();
152         ULA_starboard_precompute_weights_t.join();
153
154     }
155
156
157
158     /*=========================================================================
159     Aim: stepping motion
160     -------------------------------------------------------------------------*/
161     void step(float timestep){
162
163         // updating location
164         this->location = this->location + this->velocity * timestep;
165         if(DEBUGMODE_AUV) cout<<"\t AUVClass: page 81 \n";
166
167         // updating attributes of members
168         this->syncComponentAttributes();
169         if(DEBUGMODE_AUV) cout<<"\t AUVClass: page 85 \n";
170     }
171
172
173
174     /*=========================================================================
175     Aim: updateAttributes
176     -------------------------------------------------------------------------*/
177     void syncComponentAttributes(){
178
179         // updating ULA attributes
180         if(DEBUGMODE_AUV) cout<<"\t AUVClass: page 97 \n";
181
182         // updating locations
183         this->ULA_fls.location       = this->location;
184         this->ULA_port.location      = this->location;
185         this->ULA_starboard.location = this->location;
186
187         // updating the pointing direction of the ULAs
188         Tensor ula_fls_sensor_direction_spherical = \
189             fCart2Sph(this->pointing_direction);         // spherical coords
190         ula_fls_sensor_direction_spherical[0]          = \
191             ula_fls_sensor_direction_spherical[0] - 90;
192         Tensor ula_fls_sensor_direction_cart     = \
193             fSph2Cart(ula_fls_sensor_direction_spherical);
194
195         this->ULA_fls.sensorDirection       = ula_fls_sensor_direction_cart; // assigning sensor directionf or
                ULA-FLS
196         this->ULA_port.sensorDirection      = -this->pointing_direction;     // assigning sensor direction for
                ULA-Port
197         this->ULA_starboard.sensorDirection = -this->pointing_direction;     // assigning sensor direction for
                ULA-Starboard
198
199         // // calling the function to update the arguments
200         // this->ULA_fls.buildCoordinatesBasedOnLocation();  if(DEBUGMODE_AUV) cout<<"\t AUVClass: line 109
```

```
            \n";
201         // this->ULA_port.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) cout<<"\t AUVClass: line 111
                \n";
202         // this->ULA_starboard.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) cout<<"\t AUVClass: line
                113 \n";
203
204         // updating transmitter locations
205         this->transmitter_fls.location    = this->location;
206         this->transmitter_port.location   = this->location;
207         this->transmitter_starboard.location = this->location;
208
209         // updating transmitter pointing directions
210         this->transmitter_fls.updatePointingAngle(     this->pointing_direction);
211         this->transmitter_port.updatePointingAngle(    this->pointing_direction);
212         this->transmitter_starboard.updatePointingAngle( this->pointing_direction);
213     }
214
215
216
217     /*==========================================================================
218     Aim: operator overriding for printing
219     --------------------------------------------------------------------------*/
220     friend ostream& operator<<(ostream& os, AUVClass &auv){
221         os<<"\t location = "<<transpose(auv.location, 0, 1)<<endl;
222         os<<"\t velocity = "<<transpose(auv.velocity, 0, 1)<<endl;
223         return os;
224     }
225
226
227     /*==========================================================================
228     Aim: Subsetting Scatterers
229     --------------------------------------------------------------------------*/
230     void subsetScatterers(ScattererClass* scatterers,\
231                           TransmitterClass* transmitterObj,\
232                           float tilt_angle){
233
234         // ensuring components are synced
235         this->syncComponentAttributes();
236         if(DEBUGMODE_AUV) cout<<"\t AUVClass: page 120 \n";
237
238         // calling the method associated with the transmitter
239         if(DEBUGMODE_AUV) {cout<<"\t\t scatterers.shape = "; fPrintTensorSize(scatterers->coordinates);}
240         if(DEBUGMODE_AUV) cout<<"\t\t tilt_angle = "<<tilt_angle<<endl;
241         transmitterObj->subsetScatterers(scatterers, tilt_angle);
242         if(DEBUGMODE_AUV) cout<<"\t AUVClass: page 124 \n";
243     }
244
245     // yaw-correction matrix
246     Tensor createYawCorrectionMatrix(Tensor pointing_direction_spherical, \
247                                      float target_azimuth_deg){
248
249         // building parameters
250         Tensor azimuth_correction        = tensor({target_azimuth_deg}).to(DATATYPE).to(DEVICE) - \
251                                                 pointing_direction_spherical[0];
252         Tensor azimuth_correction_radians = azimuth_correction * PI / 180;
253
254         Tensor yawCorrectionMatrix = \
255             tensor({cos(azimuth_correction_radians).item<float>(), \
256                         cos(tensor({90}).to(DATATYPE).to(DEVICE)*PI/180 +
                                azimuth_correction_radians).item<float>(), \
257                         (float)0,                                                  \
258                         sin(azimuth_correction_radians).item<float>(), \
259                         sin(tensor({90}).to(DATATYPE).to(DEVICE)*PI/180 +
                                azimuth_correction_radians).item<float>(), \
260                         (float)0,                                                  \
261                         (float)0,                                                  \
262                         (float)0,                                                  \
263                         (float)1}).reshape({3,3}).to(DATATYPE).to(DEVICE);
264
265         // returning the matrix
266         return yawCorrectionMatrix;
267     }
268
269     // pitch-correction matrix
270     Tensor createPitchCorrectionMatrix(Tensor pointing_direction_spherical, \
```

```
                                    float target_elevation_deg){

        // building parameters
        Tensor elevation_correction        = tensor({target_elevation_deg}).to(DATATYPE).to(DEVICE) - \
                                                pointing_direction_spherical[1];
        Tensor elevation_correction_radians = elevation_correction * PI / 180;

        // creating the matrix
        Tensor pitchCorrectionMatrix = \
            tensor({{(float)1,                                                  \
                        (float)0,                                                \
                        (float)0,                                                \
                        (float)0,                                                \
                        cos(elevation_correction_radians).item<float>(), \
                        cos(tensor({90}).to(DATATYPE).to(DEVICE)*PI/180 +
                            elevation_correction_radians).item<float>(),\
                        (float)0,                                                \
                        sin(elevation_correction_radians).item<float>(), \
                        sin(tensor({90}).to(DATATYPE).to(DEVICE)*PI/180 +
                            elevation_correction_radians).item<float>()}).reshape({3,3}).to(DATATYPE);

        // returning the matrix
        return pitchCorrectionMatrix;
    }

    // Signal Simulation
    void simulateSignal(ScattererClass& scatterer){

        // printing status
        cout << "\t AUVClass::simulateSignal: Began Signal Simulation" << endl;

        // making three copies
        ScattererClass scatterer_fls    = scatterer;
        ScattererClass scatterer_port   = scatterer;
        ScattererClass scatterer_starboard = scatterer;

        // finding the pointing direction in spherical
        Tensor auv_pointing_direction_spherical = fCart2Sph(this->pointing_direction);

        // asking the transmitters to subset the scatterers by multithreading
        thread transmitterFLSSubset_t(&AUVClass::subsetScatterers, this, \
                                      &scatterer_fls,\
                                      &this->transmitter_fls, \
                                      (float)0);
        thread transmitterPortSubset_t(&AUVClass::subsetScatterers, this, \
                                       &scatterer_port,\
                                       &this->transmitter_port, \
                                       auv_pointing_direction_spherical[1].item<float>());
        thread transmitterStarboardSubset_t(&AUVClass::subsetScatterers, this, \
                                            &scatterer_starboard, \
                                            &this->transmitter_starboard, \
                                            - auv_pointing_direction_spherical[1].item<float>());

        // joining the subset threads back
        transmitterFLSSubset_t.join();
        transmitterPortSubset_t.join();
        transmitterStarboardSubset_t.join();


        // multithreading the saving tensors part.
        thread savetensor_t(fSaveSeafloorScatteres, \
                            scatterer,              \
                            scatterer_fls,          \
                            scatterer_port,         \
                            scatterer_starboard);


        // asking ULAs to simulate signal through multithreading
        thread ulafls_signalsim_t(&ULAClass::nfdc_simulateSignal,   \
                                  &this->ULA_fls,                   \
                                  &scatterer_fls,                   \
                                  &this->transmitter_fls);
        thread ulaport_signalsim_t(&ULAClass::nfdc_simulateSignal,  \
                                   &this->ULA_port,                 \
                                   &scatterer_port,                 \
```

```
                                    &this->transmitter_port);
        thread ulastarboard_signalsim_t(&ULAClass::nfdc_simulateSignal, \
                                        &this->ULA_starboard,           \
                                        &scatterer_starboard,           \
                                        &this->transmitter_starboard);

        // joining them back
        ulafls_signalsim_t.join();       // joining back the thread for ULA-FLS
        ulaport_signalsim_t.join();      // joining back the signals-sim thread for ULA-Port
        ulastarboard_signalsim_t.join(); // joining back the signal-sim thread for ULA-Starboard
        savetensor_t.join();             // joining back the signal-sim thread for tensor-saving


    }

    // Imaging Function
    /* ========================================================================
    ------------------------------------------------------------------------ */
    void image(){

        // asking ULAs to decimate the signals obtained at each time step
        thread ULA_fls_image_t(&ULAClass::nfdc_decimateSignal,      \
                               &this->ULA_fls,                      \
                               &this->transmitter_fls);
        thread ULA_port_image_t(&ULAClass::nfdc_decimateSignal,     \
                                &this->ULA_port,                    \
                                &this->transmitter_port);
        thread ULA_starboard_image_t(&ULAClass::nfdc_decimateSignal, \
                                     &this->ULA_starboard,          \
                                     &this->transmitter_starboard);

        // joining the threads back
        ULA_fls_image_t.join();
        ULA_port_image_t.join();
        ULA_starboard_image_t.join();

        // saving the decimated signal
        if (SAVE_DECIMATED_SIGNAL_MATRIX) {
            cout << "\t AUVClass::image: saving decimated signal matrix" \
                 << endl;
            save(this->ULA_fls.signalMatrix, \
                   "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/decimated_signalMatrix_fls.pt");
            save(this->ULA_port.signalMatrix, \
                      "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/decimated_signalMatrix_port.pt");
            save(this->ULA_starboard.signalMatrix, \
                      "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/decimated_signalMatrix_starboard.pt");
        }

        // asking ULAs to match-filter the signals
        thread ULA_fls_matchfilter_t(             \
            &ULAClass::nfdc_matchFilterDecimatedSignal, \
            &this->ULA_fls);
        thread ULA_port_matchfilter_t(            \
            &ULAClass::nfdc_matchFilterDecimatedSignal, \
            &this->ULA_port);
        thread ULA_starboard_matchfilter_t(       \
            &ULAClass::nfdc_matchFilterDecimatedSignal, \
            &this->ULA_starboard);

        // joining the threads back
        ULA_fls_matchfilter_t.join();
        ULA_port_matchfilter_t.join();
        ULA_starboard_matchfilter_t.join();


        // saving the decimated signal
        if (SAVE_MATCHFILTERED_SIGNAL_MATRIX) {

            // saving the tensors
            cout << "\t AUVClass::image: saving match-filtered signal matrix" \
                 << endl;
            save(this->ULA_fls.signalMatrix, \
                   "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/matchfiltered_signalMatrix_fls.pt");
            save(this->ULA_port.signalMatrix, \
                      "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/matchfiltered_signalMatrix_port.pt");
```

```
419             save(this->ULA_starboard.signalMatrix, \
420                       "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/matchfiltered_signalMatrix_starboard.pt");
421 
422             // running python-script
423 
424         }
425 
426 
427         // performing the beamforming
428         thread ULA_fls_beamforming_t(&ULAClass::nfdc_beamforming,    \
429                                      &this->ULA_fls,                    \
430                                      &this->transmitter_fls);
431         thread ULA_port_beamforming_t(&ULAClass::nfdc_beamforming,  \
432                                       &this->ULA_port,                  \
433                                       &this->transmitter_port);
434         thread ULA_starboard_beamforming_t(&ULAClass::nfdc_beamforming, \
435                                            &this->ULA_starboard,        \
436                                            &this->transmitter_starboard);
437 
438         // joining the filters back
439         ULA_fls_beamforming_t.join();
440         ULA_port_beamforming_t.join();
441         ULA_starboard_beamforming_t.join();
442 
443 
444     }
445 
446 
447 
448 
449     /* ============================================================================
450     Aim: directly create acoustic image
451     ---------------------------------------------------------------------------- */
452     void createAcousticImage(ScattererClass* scatterers){
453 
454         // making three copies
455         ScattererClass scatterer_fls     = scatterers;
456         ScattererClass scatterer_port    = scatterers;
457         ScattererClass scatterer_starboard = scatterers;
458 
459         // printing size of scatterers before subsetting
460         PRINTSMALLLINE
461         cout<< "\t > AUVClass::createAcousticImage: Beginning Scatterer Subsetting"<<endl;
462         cout<<"\t AUVClass::createAcousticImage: scatterer_fls.coordinates.shape (before) = ";
                fPrintTensorSize(scatterer_fls.coordinates);
463         cout<<"\t AUVClass::createAcousticImage: scatterer_port.coordinates.shape (before) = ";
                fPrintTensorSize(scatterer_port.coordinates);
464         cout<<"\t AUVClass::createAcousticImage: scatterer_starboard.coordinates.shape (before) = ";
                fPrintTensorSize(scatterer_starboard.coordinates);
465 
466         // finding the pointing direction in spherical
467         Tensor auv_pointing_direction_spherical = fCart2Sph(this->pointing_direction);
468 
469         // asking the transmitters to subset the scatterers by multithreading
470         thread transmitterFLSSubset_t(&AUVClass::subsetScatterers, this, \
471                                       &scatterer_fls,\
472                                       &this->transmitter_fls, \
473                                       (float)0);
474         thread transmitterPortSubset_t(&AUVClass::subsetScatterers, this, \
475                                        &scatterer_port,\
476                                        &this->transmitter_port, \
477                                        auv_pointing_direction_spherical[1].item<float>());
478         thread transmitterStarboardSubset_t(&AUVClass::subsetScatterers, this, \
479                                             &scatterer_starboard, \
480                                             &this->transmitter_starboard, \
481                                             - auv_pointing_direction_spherical[1].item<float>());
482 
483         // joining the subset threads back
484         transmitterFLSSubset_t.join(    );
485         transmitterPortSubset_t.join(   );
486         transmitterStarboardSubset_t.join( );
487 
488 
489         // asking the ULAs to directly create acoustic images
490         thread ULA_fls_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, this->ULA_fls, \
```

```
491                                         &scatterer_fls, &this->transmitter_fls);
492         thread ULA_port_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, &this->ULA_port, \
493                                         &scatterer_port, &this->transmitter_port);
494         thread ULA_starboard_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, &this->ULA_starboard, \
495                                               &scatterer_starboard, &this->transmitter_starboard);
496 
497         // joining the threads back
498         ULA_fls_acoustic_image_t.join(  );
499         ULA_port_acoustic_image_t.join( );
500         ULA_starboard_acoustic_image_t.join();
501 
502     }
503 
504 
505 };
506 
507 
508 
509 
510 
511 
512 
513 
514 
515 
516 
517 
518 
519 
520 
521 
522 
523 
524 
525 
526 
527 
528 // 0.0000,
529 // 0.0000,
530 // 0.0000,
531 // 0.0000,
532 // 0.0000,
533 // 0.0000,
534 // 0.0000,
535 // 0.0000,
536 // 0.0000,
537 // 0.0000,
538 // 0.0000,
539 // 0.0000,
540 // 0.0000,
541 // 0.0000,
542 // 0.0000,
543 // 0.0000,
544 // 0.0000,
545 // 0.0000,
546 // 0.0000,
547 // 0.0000,
548 // 0.0000,
549 // 0.0000,
550 // 0.0000,
551 // 0.0000,
552 // 0.0000,
553 // 0.0000,
554 // 0.0000,
555 // 0.0000,
556 // 0.0000,
557 // 0.0000,
558 // 0.0000,
559 // 0.0001,
560 // 0.0001,
561 // 0.0002,
562 // 0.0003,
563 // 0.0006,
564 // 0.0009,
565 // 0.0014,
```

```
566 // 0.0022, 0.0032, 0.0047, 0.0066, 0.0092, 0.0126, 0.0168, 0.0219, 0.0281, 0.0352, 0.0432, 0.0518, 0.0609,
        0.0700, 0.0786, 0.0861, 0.0921, 0.0958, 0.0969, 0.0950, 0.0903, 0.0833, 0.0755, 0.0694, 0.0693, 0.0825,
        0.1206
```

8.2 Setup Scripts

8.2.1 Seafloor Setup

Following is the script to be run to setup the seafloor.

```
/* ====================================
Aim: Setup sea floor
====================================*/

// including headerfiles
#include <torch/torch.h>
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/ScattererClass.h"

// including functions
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fCreateHills.cpp"
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fCreateBoxes.cpp"

#ifndef DEVICE
    #define DEVICE       kCPU
    // #define DEVICE       kMPS
    // #define DEVICE       kCUDA
#endif

// adding terrrain features
#define BOXES                false
#define HILLS                true
#define DEBUG_SEAFLOOR       false
#define SAVETENSORS_Seafloor false
#define PLOT_SEAFLOOR        false

// functin that setups the sea-floor
void SeafloorSetup(ScattererClass& scatterers) {

    // sea-floor bounds
    int bed_width = 100; // width of the bed (x-dimension)
    int bed_length = 100; // length of the bed (y-dimension)

    // creating some tensors to pass. This is put outside to maintain scope
    Tensor box_coordinates = zeros({3,1}).to(DATATYPE).to(DEVICE);
    Tensor box_reflectivity = zeros({1,1}).to(DATATYPE).to(DEVICE);

    // creating boxes
    if (BOXES)
        fCreateBoxes(bed_width, \
                    bed_length, \
                    box_coordinates, \
                    box_reflectivity);

    // scatter-intensity
    // int bed_width_density    = 100; // density of points along x-dimension
    // int bed_length_density   = 100; // density of points along y-dimension
    int bed_width_density    = 10; // density of points along x-dimension
    int bed_length_density   = 10; // density of points along y-dimension

    // setting up coordinates
    auto xpoints = linspace(0, \
                                bed_width, \
                                bed_width * bed_width_density).to(DEVICE);
    auto ypoints = linspace(0, \
                                bed_length, \
                                bed_length * bed_length_density).to(DEVICE);

    // creating mesh
    auto mesh_grid = meshgrid({xpoints, ypoints}, "ij");
    auto X         = mesh_grid[0];
    auto Y         = mesh_grid[1];
    X              = reshape(X, {1, X.numel()});
    Y              = reshape(Y, {1, Y.numel()});

    // creating heights of scattereres
    if(HILLS == true){
```

```
67
68          // setting up hill parameters
69          int num_hills = 10;
70
71          // setting up placement of hills
72          Tensor points2D = cat({X, Y}, 0);
73          Tensor min2D    = std::get<0>(min(points2D, 1, true));
74          Tensor max2D    = std::get<0>(max(points2D, 1, true));
75          Tensor hill_means = \
76              min2D                                       \
77              + mul(rand({2, num_hills}), \
78                          max2D - min2D);
79
80          // setting up hill dimensions
81          Tensor hill_dimensions_min = \
82              tensor({5, \
83                          5, \
84                          2}).reshape({3,1});
85          Tensor hill_dimensions_max = \
86              tensor({30, \
87                          30, \
88                          10}).reshape({3,1});
89          Tensor hill_dimensions = \
90              hill_dimensions_min + \
91              mul(hill_dimensions_max - hill_dimensions_min, \
92                        rand({3, num_hills}));
93
94          // calling the hill-creation function
95          fCreateHills(hill_means, \
96                       hill_dimensions, \
97                       points2D);
98
99          // setting up floor reflectivity
100         Tensor floorScatter_reflectivity = \
101             ones({1, Y.numel()}).to(DEVICE);
102
103         // populating the values of the incoming argument.
104         scatterers.coordinates  = points2D; // assigning coordinates
105         scatterers.reflectivity = floorScatter_reflectivity;// assigning reflectivity
106     }
107     else{
108
109         // assigning flat heights
110         Tensor Z = zeros({1, Y.numel()}).to(DEVICE);
111
112         // setting up floor coordinates
113         Tensor floorScatter_coordinates = cat({X, Y, Z}, 0);
114         Tensor floorScatter_reflectivity = ones({1, Y.numel()}).to(DEVICE);
115
116         // populating the values of the incoming argument.
117         scatterers.coordinates  = floorScatter_coordinates; // assigning coordinates
118         scatterers.reflectivity = floorScatter_reflectivity;// assigning reflectivity
119     }
120
121     // combining the values
122     if(DEBUG_SEAFLOOR) std::cout<<"\t SeafloorSetup: line 166 \n";
123     if(DEBUG_SEAFLOOR) {std::cout<<"\t scatterers.coordinates.shape = ";
            fPrintTensorSize(scatterers.coordinates);}
124     if(DEBUG_SEAFLOOR) {std::cout<<"\t box_coordinates.shape = "; fPrintTensorSize(box_coordinates);}
125     if(DEBUG_SEAFLOOR) {std::cout<<"\t scatterers.reflectivity.shape = ";
            fPrintTensorSize(scatterers.reflectivity);}
126     if(DEBUG_SEAFLOOR) {std::cout<<"\t box_reflectivity = "; fPrintTensorSize(box_reflectivity);}
127
128
129     // assigning values to the coordinates
130     scatterers.coordinates  = cat({scatterers.coordinates, box_coordinates}, 1);
131     scatterers.reflectivity = cat({scatterers.reflectivity, box_reflectivity}, 1);
132
133     // saving tensors
134     if(SAVETENSORS_Seafloor){
135         save(scatterers.coordinates, \
136                     "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt.pt");
137         std::cout<<"SeafloorSetup: Saved Seafloor "<<std::endl;
138     }
139
```

```
140 }
```

8.2.2 Transmitter Setup

Following is the script to be run to setup the transmitter.

```
/* ====================================
Aim: Setup sea floor
====================================*/
// #include <torch/torch.h>
// #include <cmath>

#ifndef DEVICE
    // #define DEVICE       torch::kMPS
    #define DEVICE       torch::kCPU
#endif



// function to calibrate the transmitters
void TransmitterSetup(TransmitterClass& transmitter_fls,
                      TransmitterClass& transmitter_port,
                      TransmitterClass& transmitter_starboard) {

    // Setting up transmitter
    float sampling_frequency = 160e3;                 // sampling frequency
    float f1                 = 50e3;                  // first frequency of LFM
    float f2                 = 70e3;                  // second frequency of LFM
    float fc                 = (f1 + f2)/2;           // finding center-frequency
    float bandwidth          = std::abs(f2 - f1); // bandwidth
    float pulselength        = 5e-2;                   // time of recording

    // building LFM
    torch::Tensor timearray = torch::linspace(0, \
                                      pulselength, \
                                      floor(pulselength * sampling_frequency)).to(DEVICE);
    float K                 = (f2 - f1)/pulselength;            // calculating frequency-slope
    torch::Tensor Signal    = K * timearray;                    // frequency at each time-step, with f1 = 0
    Signal                  = torch::mul(2*PI*(f1 + Signal), \
                                  timearray);             // creating
    Signal                  = cos(Signal);                      // calculating signal


    // Setting up transmitter
    torch::Tensor location              = torch::zeros({3,1}).to(DEVICE); // location of transmitter
    float azimuthal_angle_fls           = 0;                // initial pointing direction
    float azimuthal_angle_port          = 90;                // initial pointing direction
    float azimuthal_angle_starboard     = -90;                 // initial pointing direction

    float elevation_angle               = -60;               // initial pointing direction

    float azimuthal_beamwidth_fls       = 20;                // azimuthal beamwidth of the signal cone
    float azimuthal_beamwidth_port      = 20;                // azimuthal beamwidth of the signal cone
    float azimuthal_beamwidth_starboard = 20;                // azimuthal beamwidth of the signal cone

    float elevation_beamwidth_fls       = 20;                // elevation beamwidth of the signal cone
    float elevation_beamwidth_port      = 20;                // elevation beamwidth of the signal cone
    float elevation_beamwidth_starboard = 20;                // elevation beamwidth of the signal cone

    int azimuthQuantDensity      = 10;  // number of points, a degree is split into quantization density
        along azimuth (used for shadowing)
    int elevationQuantDensity    = 10;  // number of points, a degree is split into quantization density
        along elevation (used for shadowing)
    float rangeQuantSize         = 10;  // the length of a cell (used for shadowing)

    float azimuthShadowThreshold = 1;     // azimuth threshold    (in degrees)
    float elevationShadowThreshold = 1;   // elevation threshold  (in degrees)



    // transmitter-fls
    transmitter_fls.location               = location;           // Assigning location
    transmitter_fls.Signal                 = Signal;             // Assigning signal
    transmitter_fls.azimuthal_angle        = azimuthal_angle_fls; // assigning azimuth angle
    transmitter_fls.elevation_angle        = elevation_angle;     // assigning elevation angle
    transmitter_fls.azimuthal_beamwidth    = azimuthal_beamwidth_fls; // assigning azimuth-beamwidth
```

```
69      transmitter_fls.elevation_beamwidth    = elevation_beamwidth_fls; // assigning elevation-beamwidth
70      // updating quantization densities
71      transmitter_fls.azimuthQuantDensity    = azimuthQuantDensity;     // assigning azimuth quant density
72      transmitter_fls.elevationQuantDensity  = elevationQuantDensity;   // assigning elevation quant density
73      transmitter_fls.rangeQuantSize         = rangeQuantSize;          // assigning range-quantization
74      transmitter_fls.azimuthShadowThreshold = azimuthShadowThreshold;  // azimuth-threshold in shadowing
75      transmitter_fls.elevationShadowThreshold = elevationShadowThreshold; // elevation-threshold in shadowing
76      // signal related
77      transmitter_fls.f_low                  = f1;         // assigning lower frequency
78      transmitter_fls.f_high                 = f2;         // assigning higher frequency
79      transmitter_fls.fc                     = fc;         // assigning center frequency
80      transmitter_fls.bandwidth              = bandwidth;  // assigning bandwidth
81
82
83
84      // transmitter-portside
85      transmitter_port.location              = location;                  // Assigning location
86      transmitter_port.Signal                = Signal;                    // Assigning signal
87      transmitter_port.azimuthal_angle       = azimuthal_angle_port;      // assigning azimuth angle
88      transmitter_port.elevation_angle       = elevation_angle;           // assigning elevation angle
89      transmitter_port.azimuthal_beamwidth   = azimuthal_beamwidth_port;  // assigning azimuth-beamwidth
90      transmitter_port.elevation_beamwidth   = elevation_beamwidth_port;  // assigning elevation-beamwidth
91      // updating quantization densities
92      transmitter_port.azimuthQuantDensity   = azimuthQuantDensity;       // assigning azimuth quant density
93      transmitter_port.elevationQuantDensity = elevationQuantDensity;     // assigning elevation quant density
94      transmitter_port.rangeQuantSize        = rangeQuantSize;            // assigning range-quantization
95      transmitter_port.azimuthShadowThreshold = azimuthShadowThreshold;   // azimuth-threshold in shadowing
96      transmitter_port.elevationShadowThreshold = elevationShadowThreshold; // elevation-threshold in shadowing
97      // signal related
98      transmitter_port.f_low                 = f1;                        // assigning lower frequency
99      transmitter_port.f_high                = f2;                        // assigning higher frequency
100     transmitter_port.fc                    = fc;                        // assigning center frequency
101     transmitter_port.bandwidth             = bandwidth;                 // assigning bandwidth
102
103
104
105     // transmitter-starboard
106     transmitter_starboard.location               = location;                  // assigning location
107     transmitter_starboard.Signal                 = Signal;                    // assigning signal
108     transmitter_starboard.azimuthal_angle        = azimuthal_angle_starboard; // assigning azimuthal signal
109     transmitter_starboard.elevation_angle        = elevation_angle;
110     transmitter_starboard.azimuthal_beamwidth    = azimuthal_beamwidth_starboard;
111     transmitter_starboard.elevation_beamwidth    = elevation_beamwidth_starboard;
112     // updating quantization densities
113     transmitter_starboard.azimuthQuantDensity    = azimuthQuantDensity;
114     transmitter_starboard.elevationQuantDensity  = elevationQuantDensity;
115     transmitter_starboard.rangeQuantSize         = rangeQuantSize;
116     transmitter_starboard.azimuthShadowThreshold  = azimuthShadowThreshold;
117     transmitter_starboard.elevationShadowThreshold = elevationShadowThreshold;
118     // signal related
119     transmitter_starboard.f_low                  = f1;         // assigning lower frequency
120     transmitter_starboard.f_high                 = f2;         // assigning higher frequency
121     transmitter_starboard.fc                     = fc;         // assigning center frequency
122     transmitter_starboard.bandwidth              = bandwidth;  // assigning bandwidth
123
124 }
```

8.2.3 Uniform Linear Array

Following is the script to be run to setup the uniform linear array.

```
/* ====================================
Aim: Setup sea floor
NOAA: 50 to 100 KHz is the transmission frequency
we'll create our LFM with 50 to 70KHz
====================================*/


// Choosing device
#ifndef DEVICE
    // #define DEVICE       kMPS
    #define DEVICE       kCPU
#endif


// the coefficients for the low-pass filter.
#define LOWPASS_DECIMATE_FILTER_COEFFICIENTS 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0001, 0.0003,
     0.0006, 0.0015, 0.0030, 0.0057, 0.0100, 0.0163, 0.0251, 0.0364, 0.0501, 0.0654, 0.0814, 0.0966, 0.1093,
     0.1180, 0.1212, 0.1179, 0.1078, 0.0914, 0.0699, 0.0451, 0.0192, -0.0053, -0.0262, -0.0416, -0.0504,
     -0.0522, -0.0475, -0.0375, -0.0239, -0.0088, 0.0057, 0.0179, 0.0263, 0.0303, 0.0298, 0.0253, 0.0177,
     0.0086, -0.0008, -0.0091, -0.0153, -0.0187, -0.0191, -0.0168, -0.0123, -0.0065, -0.0004, 0.0052, 0.0095,
     0.0119, 0.0125, 0.0112, 0.0084, 0.0046, 0.0006, -0.0031, -0.0060, -0.0078, -0.0082, -0.0075, -0.0057,
     -0.0033, -0.0006, 0.0019, 0.0039, 0.0051, 0.0055, 0.0050, 0.0039, 0.0023, 0.0005, -0.0012, -0.0025,
     -0.0034, -0.0036, -0.0034, -0.0026, -0.0016, -0.0004, 0.0007, 0.0016, 0.0022, 0.0024, 0.0023, 0.0018,
     0.0011, 0.0003, -0.0004, -0.0011, -0.0015, -0.0016, -0.0015



void ULASetup(ULAClass& ula_fls,
              ULAClass& ula_port,
              ULAClass& ula_starboard) {

    // setting up ula
    int num_sensors          = 64;                        // number of sensors
    float sampling_frequency = 160e3;                     // sampling frequency
    float inter_element_spacing = 1500/(2*sampling_frequency); // space between samples
    float recording_period   = 10e-2;                        // sampling-period


    // building the direction for the sensors
    Tensor ULA_direction = tensor({-1,0,0}).reshape({3,1}).to(DATATYPE).to(DEVICE);
    ULA_direction            = ULA_direction/linalg_norm(ULA_direction, 2, 0, true, DATATYPE).to(DEVICE);
    ULA_direction            = ULA_direction * inter_element_spacing;


    // building the coordinates for the sensors
    Tensor ULA_coordinates = mul(linspace(0, num_sensors-1, num_sensors).to(DEVICE), \
                                        ULA_direction);

    // the coefficients for the decimation filter
    Tensor lowpassfiltercoefficients = tensor({LOWPASS_DECIMATE_FILTER_COEFFICIENTS}).to(DATATYPE);


    // assigning values
    ula_fls.num_sensors         = num_sensors;           // assigning number of sensors
    ula_fls.inter_element_spacing = inter_element_spacing; // assigning inter-element spacing
    ula_fls.coordinates         = ULA_coordinates;       // assigning ULA coordinates
    ula_fls.sampling_frequency  = sampling_frequency;    // assigning sampling frequencys
    ula_fls.recording_period    = recording_period;      // assigning recording period
    ula_fls.sensorDirection     = ULA_direction;         // ULA direction
    ula_fls.lowpassFilterCoefficientsForDecimation = lowpassfiltercoefficients;


    // assigning values
    ula_port.num_sensors         = num_sensors;           // assigning number of sensors
    ula_port.inter_element_spacing = inter_element_spacing; // assigning inter-element spacing
    ula_port.coordinates         = ULA_coordinates;       // assigning ULA coordinates
    ula_port.sampling_frequency  = sampling_frequency;    // assigning sampling frequencys
    ula_port.recording_period    = recording_period;      // assigning recording period
    ula_port.sensorDirection     = ULA_direction;         // ULA direction
    ula_port.lowpassFilterCoefficientsForDecimation = lowpassfiltercoefficients;
```

```
63
64
65      // assigning values
66      ula_starboard.num_sensors         = num_sensors;           // assigning number of sensors
67      ula_starboard.inter_element_spacing = inter_element_spacing; // assigning inter-element spacing
68      ula_starboard.coordinates         = ULA_coordinates;       // assigning ULA coordinates
69      ula_starboard.sampling_frequency  = sampling_frequency;    // assigning sampling frequencys
70      ula_starboard.recording_period    = recording_period;      // assigning recording period
71      ula_starboard.sensorDirection     = ULA_direction;         // ULA direction
72      ula_starboard.lowpassFilterCoefficientsForDecimation = lowpassfiltercoefficients;
73
74  }
```

8.2.4 AUV Setup

Following is the script to be run to setup the vessel.

```
/* ====================================
Aim: Setup sea floor
NOAA: 50 to 100 KHz is the transmission frequency
we'll create our LFM with 50 to 70KHz
====================================*/

#ifndef DEVICE
    #define DEVICE       torch::kMPS
    // #define DEVICE       torch::kCPU
#endif

// ========================================================
void AUVSetup(AUVClass* auv) {

    // building properties for the auv
    torch::Tensor location       = torch::tensor({0,50,30}).reshape({3,1}).to(DATATYPE).to(DEVICE); //
        starting location of AUV
    torch::Tensor velocity       = torch::tensor({5,0, 0}).reshape({3,1}).to(DATATYPE).to(DEVICE); //
        starting velocity of AUV
    torch::Tensor pointing_direction = torch::tensor({1,0, 0}).reshape({3,1}).to(DATATYPE).to(DEVICE); //
        pointing direction of AUV

    // assigning
    auv->location        = location;          // assigning location of auv
    auv->velocity        = velocity;          // assigning vector representing velocity
    auv->pointing_direction = pointing_direction; // assigning pointing direction of auv
}
```

8.3 Function Definitions

8.3.1 Cartesian Coordinates to Spherical Coordinates

```cpp
/* ====================================
Aim: Setup sea floor
======================================*/
#include <torch/torch.h>
#include <iostream>

// hash-defines
#define PI        3.14159265
#define DEBUG_Cart2Sph false

#ifndef DEVICE
    #define DEVICE       torch::kMPS
    // #define DEVICE       torch::kCPU
#endif


// bringing in functions
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fPrintTensorSize.cpp"

#pragma once

torch::Tensor fCart2Sph(torch::Tensor cartesian_vector){

    // sending argument to the device
    cartesian_vector = cartesian_vector.to(DEVICE);
    if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 26 \n";

    // splatting the point onto xy plane
    torch::Tensor xysplat = cartesian_vector.clone().to(DEVICE);
    xysplat[2] = 0;
    if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 31 \n";

    // finding splat lengths
    // torch::Tensor xysplat_lengths = torch::linalg_norm(xysplat, 2, 0, true, DATATYPE).to(DEVICE);
    torch::Tensor xysplat_lengths = torch::linalg_norm(xysplat, 2, 0, true, torch::kFloat).to(DATATYPE);
    if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 35 \n";

    // finding azimuthal and elevation angles
    torch::Tensor azimuthal_angles = torch::atan2(xysplat[1],    xysplat[0]).to(DEVICE)   * 180/PI;
    azimuthal_angles            = azimuthal_angles.reshape({1, azimuthal_angles.numel()});
    torch::Tensor elevation_angles = torch::atan2(cartesian_vector[2], xysplat_lengths).to(DEVICE) * 180/PI;
    // torch::Tensor rho_values    = torch::linalg_norm(cartesian_vector, 2, 0, true, DATATYPE).to(DEVICE);
    torch::Tensor rho_values    = torch::linalg_norm(cartesian_vector, \
                                                     2, 0, true, torch::kFloat).to(DATATYPE);
    if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 42 \n";


    // printing values for debugging
    if (DEBUG_Cart2Sph){
        std::cout<<"azimuthal_angles.shape = "; fPrintTensorSize(azimuthal_angles);
        std::cout<<"elevation_angles.shape = "; fPrintTensorSize(elevation_angles);
        std::cout<<"rho_values.shape     = "; fPrintTensorSize(rho_values);
    }
    if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 51 \n";

    // creating tensor to send back
    torch::Tensor spherical_vector = torch::cat({azimuthal_angles, \
                                            elevation_angles, \
                                            rho_values}, 0).to(DEVICE);
    if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 57 \n";

    // returning the value
    return spherical_vector;
}
```

8.3.2 Spherical Coordinates to Cartesian Coordinates

```
/* ====================================
Aim: Setup sea floor
====================================*/
#include <torch/torch.h>

#pragma once

// hash-defines
#define PI        3.14159265
#define MYDEBUGFLAG false

#ifndef DEVICE
    // #define DEVICE       torch::kMPS
    #define DEVICE       torch::kCPU
#endif


torch::Tensor fSph2Cart(torch::Tensor spherical_vector){



    // sending argument to device
    spherical_vector = spherical_vector.to(DEVICE);

    // creating cartesian vector
    torch::Tensor cartesian_vector =
         torch::zeros({3,(int)(spherical_vector.numel()/3)}).to(DATATYPE).to(DEVICE);

    // populating it
    cartesian_vector[0] = spherical_vector[2] * \
                          torch::cos(spherical_vector[1] * PI/180) * \
                          torch::cos(spherical_vector[0] * PI/180);
    cartesian_vector[1] = spherical_vector[2] * \
                          torch::cos(spherical_vector[1] * PI/180) * \
                          torch::sin(spherical_vector[0] * PI/180);
    cartesian_vector[2] = spherical_vector[2] * \
                          torch::sin(spherical_vector[1] * PI/180);

    // returning the value
    return cartesian_vector;
}
```

8.3.3 Column-Wise Convolution

```
/* ====================================
Aim: Convolving the columns of two input matrices
====================================*/
#include <ratio>
#include <stdexcept>
#include <torch/torch.h>

#pragma once

// hash-defines
#define PI        3.14159265
#define MYDEBUGFLAG false

#ifndef DEVICE
    // #define DEVICE       torch::kMPS
    #define DEVICE       torch::kCPU
#endif


void fConvolveColumns(torch::Tensor& inputMatrix, \
                      torch::Tensor& kernelMatrix){


```

```
    // printing shape
    if(MYDEBUGFLAG) std::cout<<"inputMatrix.shape =
         ["<<inputMatrix.size(0)<<","<<inputMatrix.size(1)<<std::endl;
    if(MYDEBUGFLAG) std::cout<<"kernelMatrix.shape =
         ["<<kernelMatrix.size(0)<<","<<kernelMatrix.size(1)<<std::endl;

    // ensuring the two have the same number of columns
    if (inputMatrix.size(1) != kernelMatrix.size(1)){
        throw std::runtime_error("fConvolveColumns: arguments cannot have different number of columns");
    }


    // calculating length of final result
    int final_length = inputMatrix.size(0) + kernelMatrix.size(0) - 1; if(MYDEBUGFLAG) std::cout<<"\t\t\t
         fConvolveColumns: 27"<<std::endl;

    // converting the two arguments to float since fft doesn'tw ork with halfs
    inputMatrix = inputMatrix.to(torch::kFloat);
    kernelMatrix = kernelMatrix.to(torch::kFloat);

    // calculating FFT of the two matrices
    torch::Tensor inputMatrix_FFT = torch::fft::fftn(inputMatrix, \
                                                {final_length}, \
                                                {0}); if(MYDEBUGFLAG) std::cout<<"\t\t\t fConvolveColumns:
                                                     32"<<std::endl;
    torch::Tensor kernelMatrix_FFT = torch::fft::fftn(kernelMatrix, \
                                                 {final_length}, \
                                                 {0}); if(MYDEBUGFLAG) std::cout<<"\t\t\t fConvolveColumns:
                                                      35"<<std::endl;

    // element-wise multiplying the two matrices
    torch::Tensor MulProduct = torch::mul(inputMatrix_FFT, kernelMatrix_FFT); if(MYDEBUGFLAG)
         std::cout<<"\t\t\t fConvolveColumns: 38"<<std::endl;

    // finding the inverse FFT
    torch::Tensor convolvedResult = torch::fft::ifftn(MulProduct, \
                                                 {MulProduct.size(0)}, \
                                                 {0}); if(MYDEBUGFLAG) std::cout<<"\t\t\t fConvolveColumns:
                                                      43"<<std::endl;

    // bringing them back to the pipeline datatype
    kernelMatrix = kernelMatrix.to(DATATYPE);

    // over-riding the result with the input so that we can save memory
    inputMatrix = convolvedResult.to(DATATYPE); if(MYDEBUGFLAG) std::cout<<"\t\t\t fConvolveColumns:
         46"<<std::endl;

}
```

8.3.4 Buffer 2D

```
/* ======================================
Aim: Convolving the columns of two input matrices
======================================*/
#include <stdexcept>
#include <torch/torch.h>

#pragma once

// hash-defines
#ifndef DEVICE
    // #define DEVICE       torch::kMPS
    #define DEVICE       torch::kCPU
#endif

// #define DEBUG_Buffer2D true
#define DEBUG_Buffer2D false


void fBuffer2D(torch::Tensor& inputMatrix,
               int frame_size){
```

```
21
22      // ensuring the first dimension is 1.
23      if(inputMatrix.size(0) != 1){
24          throw std::runtime_error("fBuffer2D: The first-dimension must be 1 \n");
25      }
26
27      // padding with zeros in case it is not a perfect multiple
28      if(inputMatrix.size(1)%frame_size != 0){
29          // padding with zeros
30          int numberofzeroestoadd = frame_size - (inputMatrix.size(1) % frame_size);
31          if(DEBUG_Buffer2D) {
32              std::cout  << "\t\t\t fBuffer2D: frame_size = "               << frame_size             <<
                    std::endl;
33              std::cout  << "\t\t\t fBuffer2D: inputMatrix.sizes().vec() = " << inputMatrix.sizes().vec() <<
                    std::endl;
34              std::cout  << "\t\t\t fBuffer2D: numberofzeroestoadd = "    << numberofzeroestoadd    << std::endl;
35          }
36
37          // creating zero matrix
38          torch::Tensor zeroMatrix = torch::zeros({inputMatrix.size(0), \
39                                                   numberofzeroestoadd, \
40                                                   inputMatrix.size(2)});
41          if(DEBUG_Buffer2D) std::cout<<"\t\t\t fBuffer2D: zeroMatrix.sizes() =
                "<<zeroMatrix.sizes().vec()<<std::endl;
42
43          // adding the zero matrix
44          inputMatrix = torch::cat({inputMatrix, zeroMatrix}, 1);
45          if(DEBUG_Buffer2D) std::cout<<"\t\t\t fBuffer2D: inputMatrix.sizes().vec() =
                "<<inputMatrix.sizes().vec()<<std::endl;
46      }
47
48      // calculating some parameters
49      // int num_frames = inputMatrix.size(1)/frame_size;
50      int num_frames = std::ceil(inputMatrix.size(1)/frame_size);
51      if(DEBUG_Buffer2D) std::cout << "\t\t\t fBuffer2D: inputMatrix.sizes = "<< inputMatrix.sizes().vec()<<
            std::endl;
52      if(DEBUG_Buffer2D) std::cout << "\t\t\t fBuffer2D: framesize = " << frame_size            << std::endl;
53      if(DEBUG_Buffer2D) std::cout << "\t\t\t fBuffer2D: num_frames = " << num_frames          << std::endl;
54
55      // defining target shape and size
56      std::vector<int64_t> target_shape = {num_frames,                    \
57                                           frame_size,                    \
58                                           inputMatrix.size(2)};
59      std::vector<int64_t> target_strides = {frame_size * inputMatrix.size(2), \
60                                           inputMatrix.size(2),           \
61                                           1};
62      if(DEBUG_Buffer2D) std::cout << "\t\t\t fBuffer2D: STATUS: created shape and strides"<< std::endl;
63
64      // creating the transformation
65      inputMatrix = inputMatrix.as_strided(target_shape, target_strides);
66
67  }
```

8.3.5 fAnglesToTensor

```
1   #include <torch/torch.h>
2   // function: angles to vector
3   torch::Tensor fAnglesToTensor(float azimuthal_angle,
4                                 float elevation_angle)
5   {
6     // calculating tensor
7     torch::Tensor coordinateTensor = torch::tensor({cos(elevation_angle) * cos(azimuthal_angle),
8                                                     cos(elevation_angle) * sin(azimuthal_angle),
9                                                     sin(elevation_angle)}).view({3,1});
10
11    // returning value
12    return coordinateTensor;
13  }
```

8.3.6 fCalculateCosine

```
// including headerfiles
#include <torch/torch.h>

// function to calculate cosine of two tensors
torch::Tensor fCalculateCosine(torch::Tensor inputTensor1,
                               torch::Tensor inputTensor2)
{
  // column normalizing the the two signals
  inputTensor1 = fColumnNormalize(inputTensor1);
  inputTensor2 = fColumnNormalize(inputTensor2);

  // finding their dot product
  torch::Tensor dotProduct = inputTensor1 * inputTensor2;
  torch::Tensor cosineBetweenVectors = torch::sum(dotProduct,
                                                  0,
                                                  true);

  // returning the value
  return cosineBetweenVectors;

}
```

8.4 Main Scripts

8.4.1 Signal Simulation

1.

```
/*============================================================================
Aim: Signal Simulation
----------------------------------------------------------------------------
============================================================================*/

// including
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/packages.h"
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/config.h" // hash-defines
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/classes.h" // class defs
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/setupscripts.h" // setup-scripts
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/functions.h" // functions


// main-function
int main() {

    // Ensuring no-gradients are built
    NoGradGuard no_grad;

    // Builing Sea-floor
    ScattererClass SeafloorScatter;
    thread scatterThread_t(SeafloorSetup, \
                        ref(SeafloorScatter));

    // Building ULA
    ULAClass ula_fls, ula_port, ula_starboard;
    thread ulaThread_t(ULASetup, \
                    ref(ula_fls), \
                    ref(ula_port), \
                    ref(ula_starboard));

    // Building Transmitter
    TransmitterClass transmitter_fls, transmitter_port, transmitter_starboard;
    thread transmitterThread_t(TransmitterSetup,
                            ref(transmitter_fls),
                            ref(transmitter_port),
                            ref(transmitter_starboard));

    // recombining threads
    scatterThread_t.join();  // making the scattetr population thread join back
    ulaThread_t.join();      // making the ULA population thread join back
    transmitterThread_t.join(); // making the transmitter population thread join back

    // building AUV
    AUVClass auv;               // instantiating class object
    AUVSetup(&auv);          // populating

    // attaching components to the AUV
    auv.ULA_fls              = ula_fls;              // attaching ULA-FLS to AUV
    auv.ULA_port             = ula_port;             // attaching ULA-Port to AUV
    auv.ULA_starboard        = ula_starboard;        // attaching ULA-Starboard to AUV
    auv.transmitter_fls      = transmitter_fls;      // attaching Transmitter-FLS to AUV
    auv.transmitter_port     = transmitter_port;     // attaching Transmitter-Port to AUV
    auv.transmitter_starboard = transmitter_starboard; // attaching Transmitter-Starboard to AUV

    // storing
    ScattererClass SeafloorScatter_deepcopy = SeafloorScatter;

    // pre-computing the data-structures required for processing
    auv.init();

    // mimicking movement
    int number_of_stophops = 4;
    // if (true) return 0;
    for(int i = 0; i<number_of_stophops; ++i){
```

```
66
67          // time measuring
68          auto start_time = high_resolution_clock::now();
69
70          // printing some spaces
71          PRINTSPACE; PRINTSPACE; PRINTLINE; cout<<"i = "<<i<<endl; PRINTLINE
72
73          // making the deep copy
74          ScattererClass SeafloorScatter = SeafloorScatter_deepcopy;
75
76          // signal simulation
77          auv.simulateSignal(SeafloorScatter);
78
79          // saving simulated signal
80          if (SAVETENSORS) {
81
82              // saving the signal matrix tensors
83              save(auv.ULA_fls.signalMatrix, \
84                          "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_fls.pt");
85              save(auv.ULA_port.signalMatrix, \
86                          "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_port.pt");
87              save(auv.ULA_starboard.signalMatrix, \
88                          "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_starboard.pt");
89
90              // running python script
91              string script_to_run = \
92                  "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/Python/Plot_SignalMatrix.py";
93              thread plotSignalMatrix_t(fRunSystemScriptInSeperateThread, \
94                                        script_to_run);
95              plotSignalMatrix_t.detach();
96
97          }
98
99
100         if (IMAGING_TOGGLE) {
101
102             // creating image from signals
103             auv.image();
104
105             // saving the tensors
106             if(SAVETENSORS){
107                 // saving the beamformed images
108                 save(auv.ULA_fls.beamformedImage, \
109                             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/ULA_fls_image.pt");
110                 // save(auv.ULA_port.beamformedImage, \
111                 //            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/ULA_port_image.pt");
112                 // save(auv.ULA_starboard.beamformedImage, \
113                 //
                        "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/ULA_starboard_image.pt");
114
115                 // saving cartesian image
116                 save(auv.ULA_fls.cartesianImage, \
117                             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/ULA_fls_cartesianImage.pt");
118
119                 // // running python file
120                 // system("python
                        /Users/vrsreeganesh/Documents/GitHub/AUV/Code/Python/Plot_BeamformedImage.py");
121                 system("python /Users/vrsreeganesh/Documents/GitHub/AUV/Code/Python/Plot_cartesianImage.py");
122             }
123         }
124
125
126
127         // measuring and printing time taken
128         auto end_time  = high_resolution_clock::now();
129         duration<double> time_duration = end_time - start_time;
130         PRINTDOTS; cout<<"Time taken (i = "<<i<<") = "<<time_duration.count()<<" seconds"<<endl; PRINTDOTS
131
132         // moving to next position
133         auv.step(0.5);
134
135     }
136
137
138
```

```
139
140
141
142
143   // returning
144   return 0;
145 }
```

Chapter 9

Reading

9.1 Primary Books

1.

9.2 Interesting Papers

Chapter 10

General Purpose Templated Functions

10.1 Concatenate Functions

```cpp
// input = [vector, vector],
// output = [vector]
template <std::size_t axis, typename T>
auto concatenate(const std::vector<T>&  input_vector_A,
                 const std::vector<T>&   input_vector_B) -> std::enable_if_t<axis == 1, std::vector<T> >
{
    // creating canvas vector
    auto    num_elements  {input_vector_A.size() + input_vector_B.size()};
    auto    canvas        {std::vector<T>(num_elements, (T)0) };

    // filling up the canvas
    std::copy(input_vector_A.begin(), input_vector_A.end(),
              canvas.begin());
    std::copy(input_vector_B.begin(), input_vector_B.end(),
              canvas.begin()+input_vector_A.size());

    // moving it back
    return std::move(canvas);

}
// ============================================================
// input = [vector, vector],
// output = [matrix]
template <std::size_t axis, typename T>
auto concatenate(const std::vector<T>&  input_vector_A,
                 const std::vector<T>&   input_vector_B) -> std::enable_if_t<axis == 0,
                      std::vector<std::vector<T>> >
{
    // throwing error dimensions
    if (input_vector_A.size() != input_vector_B.size())
        std::cerr << "concatenate:: incorrect dimensions \n";

    // creating canvas
    auto    canvas     {std::vector<std::vector<T>>(
        2, std::vector<T>(input_vector_A.size())
    )};

    // filling up the dimensions
    std::copy(input_vector_A.begin(), input_vector_A.end(), canvas[0].begin());
    std::copy(input_vector_B.begin(), input_vector_B.end(), canvas[1].begin());

    // moving it back
    return std::move(canvas);

}
// ============================================================
// input = [vector, vector, vector],
// output = [matrix]
```

```cpp
template <std::size_t axis, typename T>
auto concatenate(const std::vector<T>&  input_vector_A,
                 const std::vector<T>&   input_vector_B,
                 const std::vector<T>&   input_vector_C) -> std::enable_if_t<axis == 0,
                      std::vector<std::vector<T>> >
{
    // throwing error dimensions
    if (input_vector_A.size() != input_vector_B.size() ||
        input_vector_A.size() != input_vector_C.size())
        std::cerr << "concatenate:: incorrect dimensions \n";

    // creating canvas
    auto    canvas     {std::vector<std::vector<T>>(
        3, std::vector<T>(input_vector_A.size())
    )};

    // filling up the dimensions
    std::copy(input_vector_A.begin(), input_vector_A.end(), canvas[0].begin());
    std::copy(input_vector_B.begin(), input_vector_B.end(), canvas[1].begin());
    std::copy(input_vector_C.begin(), input_vector_C.end(), canvas[2].begin());

    // moving it back
    return std::move(canvas);

}
// ========================================================
// input = [matrix, vector],
// output = [matrix]
template <std::size_t axis, typename T>
auto concatenate(const std::vector<std::vector<T>>& input_matrix,
                 const std::vector<T>               input_vector) -> std::enable_if_t<axis == 0,
                      std::vector<std::vector<T>> >
{
    // creating canvas
    auto    canvas     {input_matrix};

    // adding to the canvas
    canvas.push_back(input_vector);

    // returning
    return std::move(canvas);
}
```

10.2 Data Structures

```
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) {}
};
```

10.3 Linspace

```cpp
// 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;
};
```

10.4 Max

```
template <std::size_t axis, typename T>
auto    max(const  std::vector<std::vector<T>>  input_matrix) -> std::enable_if_t<axis == 1,
     std::vector<std::vector<T>> >
{
    // setting up canvas
    auto    canvas     {std::vector<std::vector<T>>(input_matrix.size(),std::vector<T>(1))};

    // filling up the canvas
    for(auto row = 0; row < input_matrix.size(); ++row)
        canvas[row][0] = *(std::max_element(input_matrix[row].begin(), input_matrix[row].end()));

    // returning
    return std::move(canvas);
}
```

10.5 Meshgrid

```cpp
// ============================================================================
template <typename T>
auto meshgrid(const   std::vector<T>&  x,
              const    std::vector<T>&  y)
{

    // creating and filling x-grid
    std::vector<std::vector<T>> xcanvas(y.size(), std::vector<T>(x.size(), 0));
    for(auto row = 0; row < y.size(); ++row)
        std::copy(x.begin(), x.end(), xcanvas[row].begin());

    // creating and filling y-grid
    std::vector<std::vector<T>> ycanvas(y.size(), std::vector<T>(x.size(), 0));
    for(auto col = 0; col < x.size(); ++col)
        for(auto row = 0; row < y.size(); ++row)
            ycanvas[row][col] = y[row];

    // returning
    return std::move(std::pair{xcanvas, ycanvas});

}
// ============================================================================
template <typename T>
auto meshgrid(std::vector<T>&& x,
              std::vector<T>&&  y)
{

    // creating and filling x-grid
    std::vector<std::vector<T>> xcanvas(y.size(), std::vector<T>(x.size(), 0));
    for(auto row = 0; row < y.size(); ++row)
        std::copy(x.begin(), x.end(), xcanvas[row].begin());

    // creating and filling y-grid
    std::vector<std::vector<T>> ycanvas(y.size(), std::vector<T>(x.size(), 0));
    for(auto col = 0; col < x.size(); ++col)
        for(auto row = 0; row < y.size(); ++row)
            ycanvas[row][col] = y[row];

    // returning
    return std::move(std::pair{xcanvas, ycanvas});

}
```

10.6 Minimum

```
template <std::size_t axis, typename T>
auto    min(std::vector<std::vector<T>> input_matrix) -> std::enable_if_t<axis == 1,
     std::vector<std::vector<T>> >
{
    // creating canvas
    auto    canvas     {std::vector<std::vector<T>>(input_matrix.size(),std::vector<T>(1))};

    // storing the values
    for(auto row = 0; row < input_matrix.size(); ++row)
        canvas[row][0]   = *(std::min_element(input_matrix[row].begin(), input_matrix[row].end()));

    // returning the value
    return std::move(canvas);
}
```

10.7 Division

```cpp
// ========================================================
// matrix division with scalars
template <typename T>
auto operator/(const  std::vector<T>&   input_vector,
               const   T&               input_scalar)
{
    // creating canvas
    auto    canvas     {input_vector};

    // filling canvas
    std::transform(canvas.begin(), canvas.end(),
                   canvas.begin(),
                   [&input_scalar](const auto& argx){
                       return static_cast<double>(argx) / static_cast<double>(input_scalar);
                   });

    // returning value
    return std::move(canvas);
}
// ========================================================
// matrix division with scalars
template <typename T>
auto operator/=(const  std::vector<T>&   input_vector,
                const   T&               input_scalar)
{
    // creating canvas
    auto    canvas     {input_vector};

    // filling canvas
    std::transform(canvas.begin(), canvas.end(),
                   canvas.begin(),
                   [&input_scalar](const auto& argx){
                       return static_cast<double>(argx) / static_cast<double>(input_scalar);
                   });

    // returning value
    return std::move(canvas);
}
```

10.8 Addition

```
// ============================================================================
// y = vector + vector
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)
{
    // fetching dimensions
    const   auto&   num_rows_A     {a.size()};
    const   auto&   num_cols_A     {a[0].size()};
    const   auto&   num_rows_B     {b.size()};
    const   auto&   num_cols_B     {b[0].size()};

    // choosing the three different metrics
    if (num_rows_A != num_rows_B && num_cols_A != num_cols_B){
        cout << format("a.dimensions = [{},{}], b.shape = [{},{}]\n",
                       num_rows_A, num_cols_A,
                       num_rows_B, num_cols_B);
        std::cerr << "dimensions don't match\n";
    }

    // creating canvas
    auto    canvas     {std::vector<std::vector<T>>(
        std::max(num_rows_A, num_rows_B),
        std::vector<T>(std::max(num_cols_A, num_cols_B), (T)0.00)
    )};

    // performing addition
    if (num_rows_A == num_rows_B && num_cols_A == num_cols_B){
        for(auto row = 0; row < num_rows_A; ++row){
            std::transform(a[row].begin(), a[row].end(),
                           b[row].begin(),
                           canvas[row].begin(),
                           std::plus<T>());
        }
    }
    else if(num_rows_A == num_rows_B){

```

```
73          // if number of columsn are different, check if one of the cols are one
74          const  auto    min_num_cols  {std::min(num_cols_A, num_cols_B)};
75          if (min_num_cols != 1) {std::cerr<< "Operator+: unable to broadcast\n";}
76          const  auto    max_num_cols  {std::max(num_cols_A, num_cols_B)};
77
78          // using references to tag em differently
79          const  auto&   big_matrix    {num_cols_A > num_cols_B ? a : b};
80          const  auto&   small_matrix  {num_cols_A < num_cols_B ? a : b};
81
82          // Adding to canvas
83          for(auto row = 0; row < canvas.size(); ++row){
84              std::transform(big_matrix[row].begin(), big_matrix[row].end(),
85                             canvas[row].begin(),
86                             [&small_matrix,
87                              &row](const auto& argx){
88                                  return argx + small_matrix[row][0];
89                             });
90          }
91      }
92      else if(num_cols_A == num_cols_B){
93
94          // check if the smallest column-number is one
95          const  auto    min_num_rows  {std::min(num_rows_A, num_rows_B)};
96          if(min_num_rows != 1)    {std::cerr << "Operator+ : unable to broadcast\n";}
97          const  auto    max_num_rows  {std::max(num_rows_A, num_rows_B)};
98
99          // using references to differentiate the two matrices
100         const  auto&   big_matrix    {num_rows_A > num_rows_B ? a : b};
101         const  auto&   small_matrix  {num_rows_A < num_rows_B ? a : b};
102
103         // adding to canvas
104         for(auto row = 0; row < canvas.size(); ++row){
105             std::transform(big_matrix[row].begin(), big_matrix[row].end(),
106                            small_matrix[0].begin(),
107                            canvas[row].begin(),
108                            [](const auto& argx, const auto& argy){
109                             return argx + argy;
110                            });
111         }
112     }
113     else {
114         PRINTLINE PRINTLINE PRINTLINE PRINTLINE PRINTLINE
115         cout << format("check this again \n");
116     }
117
118     // returning
119     return std::move(canvas);
120 }
```

10.9 Multiplication (Element-wise)

```cpp
// 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;
}
// scalar * vector ============================================================
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;
}
// vector * vector ============================================================
template <typename T>
auto operator*(const std::vector<T>& input_vector_A,
               const std::vector<T>&  input_vector_B)
{
    // throwing error: size-desparity
    if (input_vector_A.size() != input_vector_B.size()) {std::cerr << "operator*: size disparity \n";}

    // creating canvas
    auto    canvas     {std::vector<T>(input_vector_A)};

    // element-wise multiplying
    std::transform(input_vector_A.begin(), input_vector_A.end(),
                   input_vector_B.begin(),
                   canvas.begin(),
                   [](const auto& argx, const auto& argy){
                       return argx * argy;
                   });

    // moving it back
    return std::move(canvas);
}
// 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;
}
// matrix * matrix ============================================================
template <typename T>
auto operator*(const std::vector<std::vector<T>>& A,
               const std::vector<std::vector<T>>& B) -> std::vector<std::vector<T>>
{
    // Case 1: element-wise multiplication
    if (A.size() == B.size() && A[0].size() == B[0].size()) {
        std::vector<std::vector<T>> C(A.size(), std::vector<T>(A[0].size()));
        for (std::size_t row = 0; row < A.size(); ++row) {
            std::transform(A[row].begin(), A[row].end(),
                           B[row].begin(),
                           C[row].begin(),
                           [](const auto& x, const auto& y){ return x * y; });
        }
        return C;
    }

    // Case 2: broadcast column vector
    else if (A.size() == B.size() && B[0].size() == 1) {
        std::vector<std::vector<T>> C(A.size(), std::vector<T>(A[0].size()));
        for (std::size_t row = 0; row < A.size(); ++row) {
            std::transform(A[row].begin(), A[row].end(),
                           C[row].begin(),
                           [&](const auto& x){ return x * B[row][0]; });
        }
        return C;
    }

    // case 3: when second matrix contains just one row
    // case 4: when first matrix is just one column
    // case 5: when second matrix is just one column

    // Otherwise, invalid
    else {
        throw std::runtime_error("operator* dimension mismatch");
    }
}
// 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));
```

```
146     for(int row = 0; row < finalnumrows; ++row){for(int col = 0; col < finalnumcols;
            ++col){finaloutput[row][col] = rowcolproduct(row, col);}}
147 
148     // returning
149     return finaloutput;
150 }
151 // scalar operators ============================================================
152 auto operator*(const std::complex<double> complexscalar,
153                const double               doublescalar){
154     return complexscalar * static_cast<std::complex<double>>(doublescalar);
155 }
156 auto operator*(const double               doublescalar,
157                const std::complex<double> complexscalar){
158     return complexscalar * static_cast<std::complex<double>>(doublescalar);
159 }
160 auto operator*(const std::complex<double> complexscalar,
161                const int                  scalar){
162     return complexscalar * static_cast<std::complex<double>>(scalar);
163 }
164 auto operator*(const int                  scalar,
165                const std::complex<double> complexscalar){
166     return complexscalar * static_cast<std::complex<double>>(scalar);
167 }
```

10.10 Subtraction

```cpp
// ============================================================================
// 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;
}
// ============================================================================
template <typename T>
auto operator-(const  std::vector<std::vector<T>>&  input_matrix_A,
               const   std::vector<std::vector<T>>&  input_matrix_B)
{
    // throwing error in case of dimension differences
    if (input_matrix_A.size() != input_matrix_B.size() ||
        input_matrix_A[0].size() != input_matrix_B[0].size())
        std::cerr << "operator- dimension mismatch\n";

    // setting up canvas
    auto    canvas     {std::vector<std::vector<T>>(
        input_matrix_A.size(),
        std::vector<T>(input_matrix_A[0].size())
    )};

    // subtracting values
    for(auto row = 0; row < input_matrix_B.size(); ++row)
        std::transform(input_matrix_A[row].begin(), input_matrix_A[row].end(),
                       input_matrix_B[row].begin(),
                       canvas[row].begin(),
                       [](const auto& x, const auto& y){
                           return x - y;
                       });

    // returning
    return std::move(canvas);

}
```

10.11 Printing Containers

```cpp
// 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(const std::vector<std::vector<T>> input_matrix){
    for(const auto& row: input_matrix)
        cout << format("{}\n", row);
}
template <typename T>
void fPrintMatrix(const string&                input_string,
                  const std::vector<std::vector<T>> input_matrix){
    cout << format("{} = \n", input_string);
    for(const auto& row: input_matrix)
        cout << format("{}\n", row);
}


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>
```

```
73 auto size(const std::string inputstring, std::vector<std::vector<T>> inputMatrix){
74     cout << format("{} = [{}, {}]\n", inputstring, inputMatrix.size(), inputMatrix[0].size());
75 }
```

10.12 Random Number Generation

```
// ============================================================================
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));}
```

```
73          finaloutput.push_back(std::move(temp));
74      }
75
76      // returning the finaloutput
77      return std::move(finaloutput);
78
79  }
80  // ============================================================================
81  template <typename T>
82  auto rand_complex_double(const T           argmin,
83                           const T           argmax,
84                           const vector<int>& dimensions)
85  {
86
87      // throwing an error if dimension is greater than two
88      if (dimensions.size() > 2) {std::cerr << "dimensions are too high\n";}
89
90      // creating random engine
91      static std::random_device rd; // Seed
92      static std::mt19937 gen(rd()); // Mersenne Twister generator
93      std::uniform_real_distribution<> dist(argmin, argmax);
94
95      // building the finaloutput
96      vector<vector<complex<double>>> finaloutput;
97      for(int i = 0; i<dimensions[0]; ++i){
98          vector<complex<double>> temp;
99          for(int j = 0; j<dimensions[1]; ++j) {temp.push_back(static_cast<double>(dist(gen)));}
100         finaloutput.push_back(std::move(temp));
101     }
102
103     // returning the finaloutput
104     return finaloutput;
105
106 }
```

10.13 Reshape

```cpp
// ========================================================================
// reshaping a matrix into another matrix
template <std::size_t M, std::size_t N, typename T>
auto reshape(const std::vector<std::vector<T>>& input_matrix){

    // verifying size stuff
    if (M*N != input_matrix.size() * input_matrix[0].size())
        std::cerr << "Dimensions are quite different\n";

    // creating canvas
    auto    canvas     {std::vector<std::vector<T>>(
        M, std::vector<T>(N, (T)0)
    )};

    // writing to canvas
    size_t   tid           {0};
    size_t   target_row    {0};
    size_t   target_col    {0};
    for(auto row = 0; row<input_matrix.size(); ++row){
        for(auto col = 0; col < input_matrix[0].size(); ++col){
            tid           =  row * input_matrix[0].size() + col;
            target_row    =  tid/N;
            target_col    =  tid%N;
            canvas[target_row][target_col]  =  input_matrix[row][col];
        }
    }

    // moving it back
    return std::move(canvas);
}
// ========================================================================
// reshaping a matrix into a vector
template<std::size_t M, typename T>
auto reshape(const std::vector<std::vector<T>>& input_matrix){

    // checking element-count validity
    if (M != input_matrix.size() * input_matrix[0].size())
        std::cerr << "Number of elements differ\n";

    // creating canvas
    auto    canvas     {std::vector<T>(M, 0)};

    // filling canvas
    for(auto row = 0; row < input_matrix.size(); ++row)
        for(auto col = 0; col < input_matrix[0].size(); ++col)
            canvas[row * input_matrix.size() + col] = input_matrix[row][col];

    // moving it back
    return std::move(canvas);
}

// ========================================================
// Matrix to matrix
// ========================================================
template<typename T>
auto reshape(const std::vector<std::vector<T>>& input_matrix,
             const std::size_t              M,
             const std::size_t              N){

    // checking element-count validity
    if ( M * N != input_matrix.size() * input_matrix[0].size())
        std::cerr << "Number of elements differ\n";

    // creating canvas
    auto    canvas     {std::vector<std::vector<T>>(
        M, std::vector<T>(N, (T)0)
    )};

    // writing to canvas
    size_t   tid           {0};
    size_t   target_row    {0};
    size_t   target_col    {0};
```

```
73      for(auto row = 0; row<input_matrix.size(); ++row){
74          for(auto col = 0; col < input_matrix[0].size(); ++col){
75              tid           =   row * input_matrix[0].size() + col;
76              target_row    =   tid/N;
77              target_col    =   tid%N;
78              canvas[target_row][target_col]   =   input_matrix[row][col];
79          }
80      }
81
82      // moving it back
83      return std::move(canvas);
84  }
85
86  // ========================================================
87  // converting a matrix into a vector
88  // ========================================================
89  template<typename T>
90  auto reshape(const std::vector<std::vector<T>>& input_matrix,
91               const size_t                    M){
92
93      // checking element-count validity
94      if (M != input_matrix.size() * input_matrix[0].size())
95          std::cerr << "Number of elements differ\n";
96
97      // creating canvas
98      auto    canvas     {std::vector<T>(M, 0)};
99
100     // filling canvas
101     for(auto row = 0; row < input_matrix.size(); ++row)
102         for(auto col = 0; col < input_matrix[0].size(); ++col)
103             canvas[row * input_matrix.size() + col] = input_matrix[row][col];
104
105     // moving it back
106     return std::move(canvas);
107 }
```

10.14 Transpose

```
template <typename T>
auto transpose(const std::vector<T> input_vector){

    // creating canvas
    auto    canvas     {std::vector<std::vector<T>>{
        input_vector.size(),
        std::vector<T>(1)
    }};

    // filling canvas
    for(auto i = 0; i < input_vector.size(); ++i){
        canvas[i][0]   = input_vector[i];
    }

    // moving it back
    return std::move(canvas);
}
```

10.15 CSV File-Writes

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

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

10.16 abs

```cpp
// ============================================================================
// y = abs(vector)
template <typename T>
auto abs(const std::vector<T>&  input_vector)
{
    // creating canvas
    auto    canvas     {input_vector};

    // calculating abs
    std::transform(canvas.begin(),
                   canvas.end(),
                   canvas.begin(),
                   [](auto& argx){return std::abs(argx);});

    // returning
    return std::move(canvas);
}
// ============================================================================
// y = abs(matrix)
template <typename T>
auto abs(const std::vector<std::vector<T>> input_matrix)
{
    // creating canvas
    auto    canvas     {input_matrix};

    // applying element-wise abs
    std::transform(input_matrix.begin(),
                   input_matrix.end(),
                   input_matrix.begin(),
                    [](auto& argx){return std::abs(argx);});

    // returning
    return std::move(canvas);
}
```