Autonomous Underwater Vehicle: A Surveillance Protocol

S.V. Rajendran

February 10, 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.

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 3-dimensional points. In addition to the coordinates, these points also have another attribute, we term, ”reflectivity”. It is the impulse response of a single-scatterer to a probing signal.

Sea-floors in real-life are rarely flat. They contain valleys, mountains, hills and much much more rich features. Thus, training an agent to be able to perform in the sheer different number of sea-floors involve being able to create a number of different sea-floors. Even though there are countably infinite number of variations, we shall take a structured approach to creating these variations. To that end, we start with an additive approach. The full-script for creating the sea-floor is given in section 8.2.1.

2.1 Sea “Floor”

The first addition is the sea-bottom. This is the lowest set of points that are in the point-cloud representing the total sea-floor. The most basic approach to creating these are to create a large grid of 3D points with the same height: the perfectly flat sea-floor. While this is a good place to start, we’d prefer something that looks a little bit more, “natural”. This is where we bring in the concept of rolling hills. Each hill is created as per Algorithm 1. So this becomes the lowest set of points in the overall cloud representing the matter, underwater.

Algorithm 1 Hill Creation

num_hills ← Number of Hills

2.2 Simple Structures

2.2.1 Boxes

2.2.2 Sphere

Algorithm 2 Box Creation

num_hills ← Number of Hills

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 8.2.2 and the class definition for the transmitter is given in section 8.1.2.

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

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: 8.1.2;

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 8.1.3 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 overestimate 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\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 << ",";
    }
```

```
34      std::cout << "\b]" <<std::endl;
35  }
36
37  // Scatterer Class = Scatterer Class
38  // Scatterer Class = Scatterer Class
39  // Scatterer Class = Scatterer Class
40  // Scatterer Class = Scatterer Class
41  // Scatterer Class = Scatterer Class
42  class ScattererClass{
43  public:
44
45      // public variables
46      torch::Tensor coordinates; // tensor holding coordinates [3, x]
47      torch::Tensor reflectivity; // tensor holding reflectivity [1, x]
48
49      // constructor = constructor
50      ScattererClass(torch::Tensor arg_coordinates = torch::zeros({3,1}),
51                     torch::Tensor arg_reflectivity = torch::zeros({3,1})):
52                     coordinates(arg_coordinates),
53                     reflectivity(arg_reflectivity) {}
54
55      // overloading output
56      friend std::ostream& operator<<(std::ostream& os, ScattererClass& scatterer){
57
58          // printing coordinate shape
59          os<<"\t> scatterer.coordinates.shape = ";
60          print_tensor_size(scatterer.coordinates);
61
62          // printing reflectivity shape
63          os<<"\t> scatterer.reflectivity.shape = ";
64          print_tensor_size(scatterer.reflectivity);
65
66          // returning os
67          return os;
68      }
69
70      // copy constructor from a pointer
71      ScattererClass(ScattererClass* scatterers){
72
73          // copying the values
74          this->coordinates = scatterers->coordinates;
75          this->reflectivity = scatterers->reflectivity;
76      }
77
78  };
```

8.1.2 Class: Transmitter

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

```
// header-files
#include <iostream>
#include <ostream>
#include <cmath>

// Including classes
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/ScattererClass.h"

// Including functions
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fCart2Sph.cpp"
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fPrintTensorSize.cpp"
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fSph2Cart.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

#define PI            3.14159265
#define DEBUGMODE_TRANSMITTER    false

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



// control panel
#define ENABLE_RAYTRACING            false









class TransmitterClass{
public:

    // physical/intrinsic properties
    torch::Tensor location;          // location tensor
    torch::Tensor pointing_direction; // pointing direction

    // basic parameters
    torch::Tensor Signal;     // transmitted signal (LFM)
    float azimuthal_angle;    // transmitter's azimuthal pointing direction
    float elevation_angle;    // transmitter's elevation pointing direction
    float azimuthal_beamwidth; // azimuthal beamwidth of transmitter
    float elevation_beamwidth; // elevation beamwidth of transmitter
    float range;              // a parameter used for spotlight mode.

    // transmitted signal attributes
    float f_low;              // lowest frequency of LFM
    float f_high;             // highest frequency of LFM
    float fc;                 // center frequency of LFM
    float bandwidth;          // bandwidth of LFM
```

```
67
68      // shadowing properties
69      int azimuthQuantDensity;        // quantization of angles along the azimuth
70      int elevationQuantDensity;      // quantization of angles along the elevation
71      float rangeQuantSize;           // range-cell size when shadowing
72      float azimuthShadowThreshold;   // azimuth thresholding
73      float elevationShadowThreshold; // elevation thresholding
74
75      // // shadowing related
76      // torch::Tensor checkbox;         // box indicating whether a scatter for a range-angle pair has been
            found
77      // torch::Tensor finalScatterBox;  // a 3D tensor where the third dimension represnets the vector length
78      // torch::Tensor finalReflectivityBox; // to store the reflectivity
79
80
81
82      // Constructor
83      TransmitterClass(torch::Tensor location = torch::zeros({3,1}),
84                       torch::Tensor Signal    = torch::zeros({10,1}),
85                       float azimuthal_angle   = 0,
86                       float elevation_angle   = -30,
87                       float azimuthal_beamwidth = 30,
88                       float elevation_beamwidth = 30):
89                       location(location),
90                       Signal(Signal),
91                       azimuthal_angle(azimuthal_angle),
92                       elevation_angle(elevation_angle),
93                       azimuthal_beamwidth(azimuthal_beamwidth),
94                       elevation_beamwidth(elevation_beamwidth) {}
95
96      // overloading output
97      friend std::ostream& operator<<(std::ostream& os, TransmitterClass& transmitter){
98          os<<"\t> azimuth          : "<<transmitter.azimuthal_angle <<std::endl;
99          os<<"\t> elevation        : "<<transmitter.elevation_angle <<std::endl;
100         os<<"\t> azimuthal beamwidth: "<<transmitter.azimuthal_beamwidth<<std::endl;
101         os<<"\t> elevation beamwidth: "<<transmitter.elevation_beamwidth<<std::endl;
102         PRINTSMALLLINE
103         return os;
104     }
105
106     // overloading copyign operator
107     TransmitterClass& operator=(const TransmitterClass& other){
108
109         // checking self-assignment
110         if(this==&other){
111             return *this;
112         }
113
114         // allocating memory
115         this->location           = other.location;
116         this->Signal             = other.Signal;
117         this->azimuthal_angle    = other.azimuthal_angle;
118         this->elevation_angle    = other.elevation_angle;
119         this->azimuthal_beamwidth = other.azimuthal_beamwidth;
120         this->elevation_beamwidth = other.elevation_beamwidth;
121         this->range              = other.range;
122
123         // transmitted signal attributes
124         this->f_low              = other.f_low;
125         this->f_high             = other.f_high;
126         this->fc                 = other.fc;
127         this->bandwidth          = other.bandwidth;
128
129         // shadowing properties
130         this->azimuthQuantDensity    = other.azimuthQuantDensity;
131         this->elevationQuantDensity  = other.elevationQuantDensity;
132         this->rangeQuantSize         = other.rangeQuantSize;
133         this->azimuthShadowThreshold = other.azimuthShadowThreshold;
134         this->elevationShadowThreshold = other.elevationShadowThreshold;
135
136         // this->checkbox              = other.checkbox;
137         // this->finalScatterBox       = other.finalScatterBox;
138         // this->finalReflectivityBox  = other.finalReflectivityBox;
```

```
139
140         // returning
141         return *this;
142
143     };
144
145     /*=========================================================================
146     Aim: Update pointing angle
147     -------------------------------------------------------------------------
148     Note:
149         > This function updates pointing angle based on AUV's pointing angle
150         > for now, we're assuming no roll;
151     -------------------------------------------------------------------------*/
152     void updatePointingAngle(torch::Tensor AUV_pointing_vector){
153
154         // calculate yaw and pitch
155         if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 140 \n";
156         torch::Tensor AUV_pointing_vector_spherical = fCart2Sph(AUV_pointing_vector);
157         torch::Tensor yaw                       = AUV_pointing_vector_spherical[0];
158         torch::Tensor pitch                     = AUV_pointing_vector_spherical[1];
159         if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 144 \n";
160
161         // std::cout<<"\t TransmitterClass: AUV_pointing_vector = "<<torch::transpose(AUV_pointing_vector, 0,
                1)<<std::endl;
162         // std::cout<<"\t TransmitterClass: AUV_pointing_vector_spherical =
                "<<torch::transpose(AUV_pointing_vector_spherical, 0, 1)<<std::endl;
163
164         // calculating azimuth and elevation of transmitter object
165         torch::Tensor absolute_azimuth_of_transmitter = yaw +
                torch::tensor({this->azimuthal_angle}).to(torch::kFloat).to(DEVICE);
166         torch::Tensor absolute_elevation_of_transmitter = pitch +
                torch::tensor({this->elevation_angle}).to(torch::kFloat).to(DEVICE);
167         if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 149 \n";
168
169         // std::cout<<"\t TransmitterClass: this->azimuthal_angle = "<<this->azimuthal_angle<<std::endl;
170         // std::cout<<"\t TransmitterClass: this->elevation_angle = "<<this->elevation_angle<<std::endl;
171         // std::cout<<"\t TransmitterClass: absolute_azimuth_of_transmitter =
                "<<absolute_azimuth_of_transmitter<<std::endl;
172         // std::cout<<"\t TransmitterClass: absolute_elevation_of_transmitter =
                "<<absolute_elevation_of_transmitter<<std::endl;
173
174         // converting back to Cartesian
175         torch::Tensor pointing_direction_spherical = torch::zeros({3,1}).to(torch::kFloat).to(DEVICE);
176         pointing_direction_spherical[0]         = absolute_azimuth_of_transmitter;
177         pointing_direction_spherical[1]         = absolute_elevation_of_transmitter;
178         pointing_direction_spherical[2]         = torch::tensor({1}).to(torch::kFloat).to(DEVICE);
179         if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 60 \n";
180
181         this->pointing_direction = fSph2Cart(pointing_direction_spherical);
182         if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 169 \n";
183
184     }
185
186     /*=========================================================================
187     Aim: Subsetting Scatterers inside the cone
188     .........................................................................
189     steps:
190         1. Find azimuth and range of all points.
191         2. Fint azimuth and range of current pointing vector.
192         3. Subtract azimuth and range of points from that of azimuth and range of current pointing vector
193         4. Use tilted ellipse equation to find points in the ellipse
194     -------------------------------------------------------------------------*/
195     void subsetScatterers(ScattererClass* scatterers,
196                           float tilt_angle){
197
198         // translationally change origin
199         scatterers->coordinates = scatterers->coordinates - this->location; if(DEBUGMODE_TRANSMITTER)
                std::cout<<"\t\t TransmitterClass: line 188 "<<std::endl;
200
201         /*
202         Note: I think something we can do is see if we can subset the matrices by checking coordinate values
                right away. If one of the coordinate values is x (relative coordiantes), we know for sure that
                the distance is greater than x, for sure. So, maybe that's something that we can work with
```

```
203         */
204
205         // Finding spherical coordinates of scatterers and pointing direction
206         torch::Tensor scatterers_spherical      = fCart2Sph(scatterers->coordinates);
                if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 191 "<<std::endl;
207         torch::Tensor pointing_direction_spherical = fCart2Sph(this->pointing_direction);
                if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 192 "<<std::endl;
208
209         // Calculating relative azimuths and radians
210         torch::Tensor relative_spherical = scatterers_spherical - pointing_direction_spherical;
                if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 199 "<<std::endl;
211
212         // clearing some stuff up
213         scatterers_spherical.reset();       if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line
                202 "<<std::endl;
214         pointing_direction_spherical.reset(); if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass:
                line 203 "<<std::endl;
215
216         // tensor corresponding to switch.
217         torch::Tensor tilt_angle_Tensor = torch::tensor({tilt_angle}).to(torch::kFloat).to(DEVICE);
                if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 206 "<<std::endl;
218
219         // calculating length of axes
220         torch::Tensor axis_a  = torch::tensor({this->azimuthal_beamwidth / 2}).to(torch::kFloat).to(DEVICE);
                if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 208 "<<std::endl;
221         torch::Tensor axis_b  = torch::tensor({this->elevation_beamwidth / 2}).to(torch::kFloat).to(DEVICE);
                if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 209 "<<std::endl;
222
223         // part of calculating the tilted ellipse
224         torch::Tensor xcosa   = relative_spherical[0] * torch::cos(tilt_angle_Tensor * PI/180);
225         torch::Tensor ysina   = relative_spherical[1] * torch::sin(tilt_angle_Tensor * PI/180);
226         torch::Tensor xsina   = relative_spherical[0] * torch::sin(tilt_angle_Tensor * PI/180);
227         torch::Tensor ycosa   = relative_spherical[1] * torch::cos(tilt_angle_Tensor * PI/180);
228         relative_spherical.reset(); if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 215
                "<<std::endl;
229
230         // finding points inside the tilted ellipse
231         torch::Tensor scatter_boolean = torch::div(torch::square(xcosa + ysina), torch::square(axis_a)) + \
232                                     torch::div(torch::square(xsina - ycosa), torch::square(axis_b)) <= 1;
                                        if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line
                                        221 "<<std::endl;
233
234         // clearing
235         xcosa.reset(); ysina.reset(); xsina.reset(); ycosa.reset(); if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t
                TransmitterClass: line 224 "<<std::endl;
236
237         // subsetting points within the elliptical beam
238         auto mask                = (scatter_boolean == 1); // creating a mask
239         scatterers->coordinates  = scatterers->coordinates.index({torch::indexing::Slice(), mask});
240         scatterers->reflectivity = scatterers->reflectivity.index({torch::indexing::Slice(), mask});
                if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 229 "<<std::endl;
241
242         // this is where histogram shadowing comes in (later)
243         if (ENABLE_RAYTRACING) {rangeHistogramShadowing(scatterers); std::cout<<"\t\t TransmitterClass: line
                232 "<<std::endl;}
244
245         // translating back to the points
246         scatterers->coordinates = scatterers->coordinates + this->location;
247
248     }
249
250     /*=========================================================================
251     Aim: Shadowing method (range-histogram shadowing)
252     .........................................................................
253     Note:
254         > cut down the number of threads into range-cells
255         > for each range cell, calculate histogram
256         >
257         std::cout<<"\t TransmitterClass: "
258     -------------------------------------------------------------------------*/
259     void rangeHistogramShadowing(ScattererClass* scatterers){
260
261         // converting points to spherical coordinates
```

```
262         torch::Tensor spherical_coordinates = fCart2Sph(scatterers->coordinates); std::cout<<"\t\t
                TransmitterClass: line 252 "<<std::endl;
263
264         // finding maximum range
265         torch::Tensor maxdistanceofpoints = torch::max(spherical_coordinates[2]); std::cout<<"\t\t
                TransmitterClass: line 256 "<<std::endl;
266
267         // calculating number of range-cells (verified)
268         int numrangecells = std::ceil(maxdistanceofpoints.item<int>()/this->rangeQuantSize);
269
270         // finding range-cell boundaries (verified)
271         torch::Tensor rangeBoundaries = \
272             torch::linspace(this->rangeQuantSize, \
273                             numrangecells * this->rangeQuantSize,\
274                             numrangecells); std::cout<<"\t\t TransmitterClass: line 263 "<<std::endl;
275
276         // creating the checkbox (verified)
277         int numazimuthcells   = std::ceil(this->azimuthal_beamwidth * this->azimuthQuantDensity);
278         int numelevationcells = std::ceil(this->elevation_beamwidth * this->elevationQuantDensity);
                std::cout<<"\t\t TransmitterClass: line 267 "<<std::endl;
279
280         // finding the deltas
281         float delta_azimuth  = this->azimuthal_beamwidth / numazimuthcells;
282         float delta_elevation = this->elevation_beamwidth / numelevationcells; std::cout<<"\t\t
                TransmitterClass: line 271"<<std::endl;
283
284         // creating the centers (verified)
285         torch::Tensor azimuth_centers = torch::linspace(delta_azimuth/2, \
286                                                         numazimuthcells * delta_azimuth - delta_azimuth/2, \
287                                                         numazimuthcells);
288         torch::Tensor elevation_centers = torch::linspace(delta_elevation/2, \
289                                                           numelevationcells * delta_elevation - delta_elevation/2, \
290                                                           numelevationcells); std::cout<<"\t\t TransmitterClass:
                                                                line 279"<<std::endl;
291
292         // centering (verified)
293         azimuth_centers   = azimuth_centers + torch::tensor({this->azimuthal_angle - \
294                                                              (this->azimuthal_beamwidth/2)}).to(torch::kFloat);
295         elevation_centers = elevation_centers + torch::tensor({this->elevation_angle - \
296                                                                (this->elevation_beamwidth/2)}).to(torch::kFloat);
                                                                     std::cout<<"\t\t TransmitterClass: line
                                                                     285"<<std::endl;
297
298         // building checkboxes
299         torch::Tensor checkbox          = torch::zeros({numelevationcells, numazimuthcells}, torch::kBool);
300         torch::Tensor finalScatterBox   = torch::zeros({numelevationcells, numazimuthcells,
                3}).to(torch::kFloat);
301         torch::Tensor finalReflectivityBox = torch::zeros({numelevationcells,
                numazimuthcells}).to(torch::kFloat); std::cout<<"\t\t TransmitterClass: line 290"<<std::endl;
302
303         // going through each-range-cell
304         for(int i = 0; i<(int)rangeBoundaries.numel(); ++i){
305             this->internal_subsetCurrentRangeCell(rangeBoundaries[i], \
306                                                   scatterers,           \
307                                                   checkbox,             \
308                                                   finalScatterBox,      \
309                                                   finalReflectivityBox, \
310                                                   azimuth_centers,      \
311                                                   elevation_centers,    \
312                                                   spherical_coordinates); std::cout<<"\t\t TransmitterClass: line
                                                        301"<<std::endl;
313
314             // after each-range-cell
315             torch::Tensor checkboxfilled = torch::sum(checkbox);
316             std::cout<<"\t\t\t\t checkbox-filled = "<<checkboxfilled.item<int>()<<"/"<<checkbox.numel()<<" |
                    percent = "<<100 * checkboxfilled.item<float>()/(float)checkbox.numel()<<std::endl;
317
318         }
319
320         // converting from box structure to [3, num-points] structure
321         torch::Tensor final_coords_spherical = \
322             torch::permute(finalScatterBox, {2, 0, 1}).reshape({3, (int)(finalScatterBox.numel()/3)});
323         torch::Tensor final_coords_cart = fSph2Cart(final_coords_spherical); std::cout<<"\t\t
```

```
            TransmitterClass: line 308"<<std::endl;
324         std::cout<<"\t\t finalReflectivityBox.shape = "; fPrintTensorSize(finalReflectivityBox);
325         torch::Tensor final_reflectivity = finalReflectivityBox.reshape({finalReflectivityBox.numel()});
                std::cout<<"\t\t TransmitterClass: line 310"<<std::endl;
326         torch::Tensor test_checkbox = checkbox.reshape({checkbox.numel()}); std::cout<<"\t\t TransmitterClass:
                line 311"<<std::endl;
327
328         // just taking the points corresponding to the filled. Else, there's gonna be a lot of zero zero zero
                tensors
329         auto mask = (test_checkbox == 1); std::cout<<"\t\t TransmitterClass: line 319"<<std::endl;
330         final_coords_cart = final_coords_cart.index({torch::indexing::Slice(), mask}); std::cout<<"\t\t
                TransmitterClass: line 320"<<std::endl;
331         final_reflectivity = final_reflectivity.index({mask}); std::cout<<"\t\t TransmitterClass: line
                321"<<std::endl;
332
333         // overwriting the scatterers
334         scatterers->coordinates  = final_coords_cart;
335         scatterers->reflectivity = final_reflectivity; std::cout<<"\t\t TransmitterClass: line 324"<<std::endl;
336
337     }
338
339
340     void internal_subsetCurrentRangeCell(torch::Tensor rangeupperlimit, \
341                                          ScattererClass* scatterers,        \
342                                          torch::Tensor& checkbox,           \
343                                          torch::Tensor& finalScatterBox,    \
344                                          torch::Tensor& finalReflectivityBox, \
345                                          torch::Tensor& azimuth_centers,    \
346                                          torch::Tensor& elevation_centers,  \
347                                          torch::Tensor& spherical_coordinates){
348
349         // finding indices for points in the current range-cell
350         torch::Tensor pointsincurrentrangecell = \
351             torch::mul((spherical_coordinates[2] <= rangeupperlimit) , \
352                        (spherical_coordinates[2] > rangeupperlimit - this->rangeQuantSize));
353
354         // checking out if there are no points in this range-cell
355         int num311 = torch::sum(pointsincurrentrangecell).item<int>();
356         if(num311 == 0) return;
357
358         // calculating delta values
359         float delta_azimuth  = azimuth_centers[1].item<float>() - azimuth_centers[0].item<float>();
360         float delta_elevation = elevation_centers[1].item<float>() - elevation_centers[0].item<float>();
361
362         // subsetting points in the current range-cell
363         auto mask                                  = (pointsincurrentrangecell == 1); // creating a mask
364         torch::Tensor reflectivityincurrentrangecell =
                scatterers->reflectivity.index({torch::indexing::Slice(), mask});
365         pointsincurrentrangecell                   = spherical_coordinates.index({torch::indexing::Slice(),
                mask});
366
367         // finding number of azimuth sizes and what not
368         int numazimuthcells   = azimuth_centers.numel();
369         int numelevationcells = elevation_centers.numel();
370
371         // go through all the combinations
372         for(int azi_index = 0; azi_index < numazimuthcells; ++azi_index){
373             for(int ele_index = 0; ele_index < numelevationcells; ++ele_index){
374
375                 // check if this particular azimuth-elevation direction has been taken-care of.
376                 if (checkbox[ele_index][azi_index].item<bool>()) break;
377
378                 // init (verified)
379                 torch::Tensor current_azimuth = azimuth_centers.index({azi_index});
380                 torch::Tensor current_elevation = elevation_centers.index({ele_index});
381
382                 // // finding azimuth boolean
383                 // torch::Tensor azi_neighbours = torch::abs(pointsincurrentrangecell[0] - current_azimuth);
384                 // azi_neighbours               = azi_neighbours <= delta_azimuth; // tinker with this.
385
386                 // // finding elevation boolean
387                 // torch::Tensor ele_neighbours = torch::abs(pointsincurrentrangecell[1] - current_elevation);
388                 // ele_neighbours               = ele_neighbours <= delta_elevation;
```

```
389
390             // finding azimuth boolean
391             torch::Tensor azi_neighbours = torch::abs(pointsincurrentrangecell[0] - current_azimuth);
392             azi_neighbours               = azi_neighbours <= this->azimuthShadowThreshold; // tinker with
                    this.
393
394             // finding elevation boolean
395             torch::Tensor ele_neighbours = torch::abs(pointsincurrentrangecell[1] - current_elevation);
396             ele_neighbours               = ele_neighbours <= this->elevationShadowThreshold;
397
398
399             // combining booleans: means find all points that are within the limits of both the azimuth and
                    boolean.
400             torch::Tensor neighbours_boolean = torch::mul(azi_neighbours, ele_neighbours);
401
402             // checking if there are any points along this direction
403             int num347 = torch::sum(neighbours_boolean).item<int>();
404             if (num347 == 0) continue;
405
406             // findings point along this direction
407             mask                                    = (neighbours_boolean == 1);
408             torch::Tensor coords_along_aziele_spherical =
                    pointsincurrentrangecell.index({torch::indexing::Slice(), mask});
409             torch::Tensor reflectivity_along_aziele =
                    reflectivityincurrentrangecell.index({torch::indexing::Slice(), mask});
410
411             // finding the index where the points are at the maximum distance
412             int index_where_min_range_is = torch::argmin(coords_along_aziele_spherical[2]).item<int>();
413             torch::Tensor closest_coord = coords_along_aziele_spherical.index({torch::indexing::Slice(), \
414                                                                                index_where_min_range_is});
415             torch::Tensor closest_reflectivity = reflectivity_along_aziele.index({torch::indexing::Slice(),
                    \
416                                                                                   index_where_min_range_is});
417
418             // filling the matrices up
419             finalScatterBox.index_put_({ele_index, azi_index, torch::indexing::Slice()}, \
420                                        closest_coord.reshape({1,1,3}));
421             finalReflectivityBox.index_put_({ele_index, azi_index}, \
422                                             closest_reflectivity);
423             checkbox.index_put_({ele_index, azi_index}, \
424                                 true);
425
426         }
427     }
428 }
429
430
431
432
433 };
```

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 <stdexcept>
#include <torch/torch.h>


// 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 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
    int decimation_factor;
    torch::Tensor lowpassFilterCoefficientsForDecimation; //

    // imaging related
    float range_resolution;         // theoretical range-resolution = $\frac{c}{2B}$
    float azimuthal_resolution;     // theoretical azimuth-resolution =
        $\frac{\lambda}{(N-1)*inter-element-distance}$
```

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

```
135         this->nfdc_CreateMatchFilter(transmitterObj);
136     }
137 
138     // Create match-filter
139     void nfdc_CreateMatchFilter(TransmitterClass* transmitterObj){
140 
141         // creating matrix for basebanding the signal
142         torch::Tensor basebanding_vector = \
143             torch::linspace(0, \
144                         transmitterObj->Signal.numel() - 1, \
145                         transmitterObj->Signal.numel()).reshape(transmitterObj->Signal.sizes());
146         basebanding_vector = \
147             torch::exp(COMPLEX_1j * 2 * PI * transmitterObj->fc * basebanding_vector);
148 
149         // multiplying the signal with the basebanding vector
150         torch::Tensor match_filter = \
151             torch::mul(transmitterObj->Signal, basebanding_vector);
152 
153         // low-pass filtering
154         fConvolve1D(match_filter, this->lowpassFilterCoefficientsForDecimation);
155 
156         // creating sampling-indices
157         int decimation_factor = std::floor((static_cast<float>(this->sampling_frequency)/2) / \
158                                         (static_cast<float>(transmitterObj->bandwidth)/2));
159         int final_num_samples =
              std::ceil(static_cast<float>(match_filter.numel())/static_cast<float>(decimation_factor));
160         torch::Tensor sampling_indices = \
161             torch::linspace(1, \
162                         (final_num_samples-1) * decimation_factor,
163                         final_num_samples).to(torch::kInt) - torch::tensor({1}).to(torch::kInt);
164 
165         // sampling the signal
166         match_filter = match_filter.index({sampling_indices});
167 
168         // taking conjugate and flipping the signal
169         match_filter = torch::flipud( match_filter);
170         match_filter = torch::conj(  match_filter);
171 
172         // storing the match-filter to the class member
173         this->matchFilter = match_filter;
174     }
175 
176     // overloading the "=" operator
177     ULAClass& operator=(const ULAClass& other){
178         // checking if copying to the same object
179         if(this == &other){
180             return *this;
181         }
182 
183         // copying everything
184         this->num_sensors        = other.num_sensors;
185         this->inter_element_spacing = other.inter_element_spacing;
186         this->coordinates        = other.coordinates.clone();
187         this->sampling_frequency = other.sampling_frequency;
188         this->recording_period   = other.recording_period;
189         this->sensorDirection    = other.sensorDirection.clone();
190 
191         // new additions
192         // this->location           = other.location;
193         this->lowpassFilterCoefficientsForDecimation = other.lowpassFilterCoefficientsForDecimation;
194         // this->sensorDirection    = other.sensorDirection.clone();
195         // this->signalMatrix       = other.signalMatrix.clone();
196 
197 
198         // returning
199         return *this;
200     }
201 
202     // build sensor-coordinates based on location
203     void buildCoordinatesBasedOnLocation(){
204 
205         // length-normalize the sensor-direction
206         this->sensorDirection = torch::div(this->sensorDirection, torch::linalg_norm(this->sensorDirection, \
```

```
                                                                2, 0, true, \
                                                                torch::kFloat));
    if(DEBUG_ULA) std::cout<<"\t ULAClass: line 105 \n";

    // multiply with inter-element distance
    this->sensorDirection = this->sensorDirection * this->inter_element_spacing;
    this->sensorDirection = this->sensorDirection.reshape({this->sensorDirection.numel(), 1});
    if(DEBUG_ULA) std::cout<<"\t ULAClass: line 110 \n";

    // create integer-array
    // torch::Tensor integer_array = torch::linspace(0, \
    //                                     this->num_sensors-1, \
    //                                     this->num_sensors).reshape({1,
        this->num_sensors}).to(torch::kFloat);
    torch::Tensor integer_array = torch::linspace(0,                                    \
                                        this->num_sensors-1,                            \
                                        this->num_sensors).reshape({1,                  \
                                                                this->num_sensors});
    std::cout<<"integer_array = "; fPrintTensorSize(integer_array);
    if(DEBUG_ULA) std::cout<<"\t ULAClass: line 116 \n";

    //
    torch::Tensor test = torch::mul(torch::tile(integer_array, {3, 1}).to(torch::kFloat), \
                            torch::tile(this->sensorDirection, {1,
                                this->num_sensors}).to(torch::kFloat));
    this->coordinates = this->location + test;
    if(DEBUG_ULA) std::cout<<"\t ULAClass: line 120 \n";

}

// signal simulation for the current sensor-array
void nfdc_simulateSignal(ScattererClass* scatterers,
                      TransmitterClass* transmitterObj){

    // creating signal matrix
    int numsamples    = std::ceil((this->sampling_frequency * this->recording_period));
    this->signalMatrix = torch::zeros({numsamples, this->num_sensors}).to(torch::kFloat);

    // getting shape of coordinates
    std::vector<int64_t> scatterers_coordinates_shape = scatterers->coordinates.sizes().vec();

    // making a slot out of the coordinates
    torch::Tensor slottedCoordinates = \
        torch::permute(scatterers->coordinates.reshape({scatterers_coordinates_shape[0], \
                                            scatterers_coordinates_shape[1], \
                                            1                                }), \
                    {2, 1, 0}).reshape({1, (int)(scatterers->coordinates.numel()/3), 3});

    // repeating along the y-direction number of sensor times.
    slottedCoordinates = torch::tile(slottedCoordinates, {this->num_sensors, 1, 1});
    std::vector<int64_t> slottedCoordinates_shape = slottedCoordinates.sizes().vec();

    // finding the shape of the sensor-coordinates
    std::vector<int64_t> sensor_coordinates_shape = this->coordinates.sizes().vec();

    // creating a slot tensor out of the sensor-coordinates
    torch::Tensor slottedSensors = \
        torch::permute(this->coordinates.reshape({sensor_coordinates_shape[0], \
                                        sensor_coordinates_shape[1], \
                                        1}), {2, 1, 0}).reshape({(int)(this->coordinates.numel()/3),
                                            \
                                                        1, \
                                                        3});

    // repeating slices along the y-coordinates
    slottedSensors = torch::tile(slottedSensors, {1, slottedCoordinates_shape[1], 1});

    // slotting the coordinate of the transmitter
    torch::Tensor slotted_location = torch::permute(this->location.reshape({3, 1, 1}), \
                                            {2, 1, 0}).reshape({1,1,3});
    slotted_location = torch::tile(slotted_location, \
                                {slottedCoordinates_shape[0], slottedCoordinates_shape[1], 1});

```

```
277         // subtracting to find the relative distances
278         torch::Tensor distBetweenScatterersAndSensors = \
279             torch::linalg_norm(slottedCoordinates - slottedSensors, 2, 2, true, torch::kFloat);
280 
281         // substracting distance between relative fields
282         torch::Tensor distBetweenScatterersAndTransmitter = \
283             torch::linalg_norm(slottedCoordinates - slotted_location, 2, 2, true, torch::kFloat);
284 
285         // adding up the distances
286         torch::Tensor distOfFlight   = distBetweenScatterersAndSensors + distBetweenScatterersAndTransmitter;
287         torch::Tensor timeOfFlight   = distOfFlight/1500;
288         torch::Tensor samplesOfFlight = torch::floor(timeOfFlight.squeeze() * this->sampling_frequency);
289 
290         // Adding pulses
291         for(int sensor_index = 0; sensor_index < this->num_sensors; ++sensor_index){
292             for(int scatter_index = 0; scatter_index < samplesOfFlight[0].numel(); ++scatter_index){
293 
294                 // getting the sample where the current scatter's contribution must be placed.
295                 int where_to_place = samplesOfFlight.index({sensor_index, scatter_index}).item<int>();
296 
297                 // checking whether that point is out of bounds
298                 if(where_to_place >= numsamples) continue;
299 
300                 // placing a point in there
301                 this->signalMatrix.index_put_({where_to_place, sensor_index}, \
302                                               this->signalMatrix.index({where_to_place, sensor_index}) + \
303                                               torch::tensor({1}).to(torch::kFloat));
304 
305                 this->signalMatrix.index_put_({where_to_place, sensor_index}, \
306                                               this->signalMatrix.index({where_to_place, sensor_index}) + \
307                                                 scatterers->reflectivity.index({0, scatter_index}) );
308             }
309         }
310 
311         // Convolving signals with transmitted signal
312         torch::Tensor signalTensorAsArgument = \
313             transmitterObj->Signal.reshape({transmitterObj->Signal.numel(),1});
314         signalTensorAsArgument = torch::tile(signalTensorAsArgument, \
315                                              {1, this->signalMatrix.size(1)});
316 
317         // convolving the pulse-matrix with the signal matrix
318         fConvolveColumns(this->signalMatrix,    \
319                          signalTensorAsArgument);
320 
321         // trimming the convolved signal since the signal matrix length remains the same
322         this->signalMatrix = this->signalMatrix.index({torch::indexing::Slice(0, numsamples), \
323                                                        torch::indexing::Slice()});
324 
325         // printing the shape
326         if(DEBUG_ULA) {
327             std::cout<<"\t\t\t> this->signalMatrix.shape (after signal sim) = ";
328             fPrintTensorSize(this->signalMatrix);
329         }
330 
331         return;
332     }
333 
334     // decimating the obtained signal
335     void nfdc_decimateSignal(TransmitterClass* transmitterObj){
336 
337         // creating the matrix for frequency-shifting
338         torch::Tensor integerArray = torch::linspace(0, \
339                                                      this->signalMatrix.size(0)-1, \
340                                                      this->signalMatrix.size(0)).reshape({this->signalMatrix.size(0),
                                                          1});
341         integerArray             = torch::tile(integerArray, {1, this->num_sensors});
342         integerArray             = torch::exp(COMPLEX_1j * transmitterObj->fc * integerArray);
343 
344         // storing original number of samples
345         int original_signalMatrix_numsamples = this->signalMatrix.size(0);
346 
347         // // printing
348         // std::cout << "this->signalMatrix.shape = "<< this->signalMatrix.sizes().vec() << std::endl;
```

```
349         // std::cout << "integerArray.shape      = "<< integerArray.sizes().vec()     << std::endl;
350
351         // producing frequency-shifting
352         this->signalMatrix       = torch::mul(this->signalMatrix, integerArray);
353
354         // low-pass filter
355         torch::Tensor lowpassfilter_impulseresponse = \
356             this->lowpassFilterCoefficientsForDecimation.reshape(\
357                 {this->lowpassFilterCoefficientsForDecimation.numel(), 1});
358         lowpassfilter_impulseresponse = torch::tile(lowpassfilter_impulseresponse, \
359                                 {1, this->signalMatrix.size(1)});
360
361         // Convolving
362         fConvolveColumns(this->signalMatrix, lowpassfilter_impulseresponse);
363
364         // Cutting down the extra-samples from convolution
365         this->signalMatrix = \
366             this->signalMatrix.index({torch::indexing::Slice(0, original_signalMatrix_numsamples), \
367                                 torch::indexing::Slice()});
368
369         // building parameters for downsampling
370         int decimation_factor        = std::floor(this->sampling_frequency/transmitterObj->bandwidth);
371         this->decimation_factor      = decimation_factor;
372         int numsamples_after_decimation = std::floor(this->signalMatrix.size(0)/decimation_factor);
373
374         // building the samples which will be subsetted
375         torch::Tensor samplingIndices = \
376             torch::linspace(0, \
377                         numsamples_after_decimation * decimation_factor - 1, \
378                         numsamples_after_decimation).to(torch::kInt);
379
380         // downsampling the low-pass filtered signal
381         this->signalMatrix = \
382             this->signalMatrix.index({samplingIndices, \
383                                 torch::indexing::Slice()});
384
385
386     }
387
388     /* ============================================================================
389     Aim: Match-filtering
390     ---------------------------------------------------------------------------- */
391     void nfdc_matchFilterDecimatedSignal(){
392         // Creating a 2D marix out of the signal
393         torch::Tensor matchFilter2DMatrix = \
394             this->matchFilter.reshape({this->matchFilter.numel(), 1});
395         matchFilter2DMatrix = torch::tile(matchFilter2DMatrix, \
396                                     {1, this->num_sensors});
397
398         // 2D convolving to produce the match-filtering
399         fConvolveColumns(this->signalMatrix, \
400                         matchFilter2DMatrix);
401
402     }
403
404     /* ============================================================================
405     Aim: precompute delay-matrices
406     ---------------------------------------------------------------------------- */
407     void nfdc_precomputeDelayMatrices(TransmitterClass* transmitterObj){
408
409         // calculating range-related parameters
410         int number_of_range_cells    = \
411             std::ceil(((this->recording_period * 1500)/2)/this->range_cell_size);
412         int number_of_azimuths_to_image = \
413             std::ceil(transmitterObj->azimuthal_beamwidth / this->azimuth_cell_size);
414
415         // creating centers of range-cell centers
416         torch::Tensor range_centers = \
417             this->range_cell_size *                                 \
418             torch::linspace(0,                                      \
419                         number_of_range_cells-1,                    \
420                         number_of_range_cells).to(torch::kFloat) + \
421             this->range_cell_size/2;
```

```
422         this->range_centers = range_centers;
423
424         // creating discretized azimuth-centers
425         torch::Tensor azimuth_centers = \
426             this->azimuth_cell_size *                       \
427             torch::linspace(0,                              \
428                             number_of_azimuths_to_image - 1, \
429                             number_of_azimuths_to_image) +   \
430             this->azimuth_cell_size/2;
431         this->azimuth_centers = azimuth_centers;
432
433         // finding the mesh values
434         auto range_azimuth_meshgrid = \
435             torch::meshgrid({range_centers, azimuth_centers}, "ij");
436         torch::Tensor range_grid = range_azimuth_meshgrid[0]; // the columns are range_centers
437         torch::Tensor azimuth_grid = range_azimuth_meshgrid[1]; // the rows are azimuth-centers
438
439         // going from 2D to 3D
440         range_grid = torch::tile(range_grid.reshape({range_grid.size(0), \
441                                                      range_grid.size(1), \
442                                                      1}), \
443                                  {1,1,this->num_sensors});
444         azimuth_grid = torch::tile(azimuth_grid.reshape({azimuth_grid.size(0), \
445                                                          azimuth_grid.size(1), \
446                                                          1}), \
447                                    {1, 1, this->num_sensors});
448
449         // creating x_m tensor
450         torch::Tensor sensorCoordinatesSlot = \
451             this->inter_element_spacing * \
452             torch::linspace(0, \
453                             this->num_sensors - 1,\
454                             this->num_sensors).reshape({1,1,this->num_sensors}).to(torch::kFloat);
455         sensorCoordinatesSlot = \
456             torch::tile(sensorCoordinatesSlot, \
457                         {range_grid.size(0),\
458                          range_grid.size(1),
459                          1});
460         if(DEBUG_ULA)
461             std::cout << "\t sensorCoordinatesSlot.sizes().vec() = " \
462                       << sensorCoordinatesSlot.sizes().vec()      \
463                       << std::endl;
464
465         // calculating distances
466         torch::Tensor distanceMatrix =                              \
467             torch::square(range_grid - sensorCoordinatesSlot) +     \
468             torch::mul((2 * sensorCoordinatesSlot),                 \
469                        torch::mul(range_grid,                       \
470                                   1 - torch::cos(azimuth_grid * PI/180)));
471         distanceMatrix = torch::sqrt(distanceMatrix);
472
473         // finding the time taken
474         torch::Tensor timeMatrix = distanceMatrix/1500;
475         torch::Tensor sampleMatrix = timeMatrix * this->sampling_frequency;
476
477         // finding the delay to be given
478         auto bruh390             = torch::max(sampleMatrix, 2, true);
479         torch::Tensor max_delay  = std::get<0>(bruh390);
480         torch::Tensor delayMatrix = max_delay - sampleMatrix;
481
482         // now that we have the delay entries, we need to create the matrix that does it
483         int decimation_factor = \
484             std::floor(static_cast<float>(this->sampling_frequency)/transmitterObj->bandwidth);
485         this->decimation_factor = decimation_factor;
486
487
488         // calculating frame-size
489         int frame_size = \
490             std::ceil(static_cast<float>((2 * this->range_cell_size / 1500) * \
491                       static_cast<float>(this->sampling_frequency)/decimation_factor));
492         this->frame_size = frame_size;
493
494         // calculating the buffer-zeros to add
```

```
495         int num_buffer_zeros_per_frame = \
496             static_cast<float>(this->num_sensors - 1) * \
497             static_cast<float>(this->inter_element_spacing) * \
498             this->sampling_frequency /1500;
499 
500         // storing to class member
501         this->num_buffer_zeros_per_frame = \
502             num_buffer_zeros_per_frame;
503 
504         // calculating the total frame-size
505         int total_frame_size = \
506             this->frame_size + this->num_buffer_zeros_per_frame;
507 
508         // creating the multiplication matrix
509         torch::Tensor mulFFTMatrix = \
510             torch::linspace(0, \
511                             total_frame_size-1, \
512                             total_frame_size).reshape({1, \
513                                                        total_frame_size, \
514                                                        1}).to(torch::kFloat); // creating an array
                                                             1,...,frame_size of shape [1,frame_size, 1];
515         mulFFTMatrix = \
516             torch::div(mulFFTMatrix, \
517                        torch::tensor(total_frame_size).to(torch::kFloat)); // dividing by N
518         mulFFTMatrix = mulFFTMatrix * 2 * PI * -1 * COMPLEX_1j; // creating tenosr values for -1j * 2pi * k/N
519         mulFFTMatrix = \
520             torch::tile(mulFFTMatrix, \
521                         {number_of_range_cells * number_of_azimuths_to_image, \
522                          1, \
523                          this->num_sensors}); // creating the larger tensor for it
524 
525 
526         // populating the matrix
527         for(int azimuth_index = 0; \
528             azimuth_index<number_of_azimuths_to_image; \
529             ++azimuth_index){
530             for(int range_index = 0; \
531                 range_index < number_of_range_cells; \
532                 ++range_index){
533                 // finding the delays for sensors
534                 torch::Tensor currentSensorDelays = \
535                     delayMatrix.index({range_index, \
536                                        azimuth_index, \
537                                        torch::indexing::Slice()});
538                 // reshaping it to the target size
539                 currentSensorDelays = \
540                     currentSensorDelays.reshape({1, \
541                                                  1, \
542                                                  this->num_sensors});
543                 // tiling across the plane
544                 currentSensorDelays = \
545                     torch::tile(currentSensorDelays, \
546                                 {1, total_frame_size, 1});
547                 // multiplying across the appropriate plane
548                 int index_to_place_at = \
549                     azimuth_index * number_of_range_cells + \
550                     range_index;
551                 mulFFTMatrix.index_put_({index_to_place_at, \
552                                          torch::indexing::Slice(),
553                                          torch::indexing::Slice()}, \
554                                          currentSensorDelays);
555             }
556         }
557 
558         // storing the mulFFTMatrix
559         this->mulFFTMatrix = mulFFTMatrix;
560     }
561 
562     // beamforming the signal
563     void nfdc_beamforming(TransmitterClass* transmitterObj){
564 
565         // ensuring the signal matrix is in the shape we want
566         if(this->signalMatrix.size(1) != this->num_sensors)
```

```
567             throw std::runtime_error("The second dimension doesn't correspond to the number of sensors \n");
568
569         // adding the batch-dimension
570         /* This is to accomodate a particular property of torch library.
571         In torch, the first dimension is always the batch-related dimension.
572         So in order to use the function torch::bmm(), we need to ensure that the first dimension is that of
                shape. */
573         this->signalMatrix = \
574             this->signalMatrix.reshape({1, \
575                                         this->signalMatrix.size(0), \
576                                         this->signalMatrix.size(1)});
577
578         // zero-padding to ensure correctness
579         int ideal_length                          = std::ceil(this->range_centers.numel() * this->frame_size);
580         int num_zeros_to_pad_signal_along_dimension_0 = ideal_length - this->signalMatrix.size(1);
581
582         // printing
583         PRINTSMALLLINE
584         std::cout<<"\t\t ULAClass::nfdc_beamforming | this->range_centers.numel()      =
                "<<this->range_centers.numel() <<std::endl;
585         std::cout<<"\t\t ULAClass::nfdc_beamforming | this->frame_size                = "<<this->frame_size
                        <<std::endl;
586         std::cout<<"\t\t ULAClass::nfdc_beamforming | ideal_length                    = "<<ideal_length
                            <<std::endl;
587         std::cout<<"\t\t ULAClass::nfdc_beamforming | this->signalMatrix.size(1)      =
                "<<this->signalMatrix.size(1)   <<std::endl;
588         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;
589
590         // appending or slicing based on the requirements
591         if (num_zeros_to_pad_signal_along_dimension_0 <= 0) {
592
593             // sending out a warning that slicing is going on
594             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;
595
596             // slicing the signal matrix
597             PRINTSPACE
598             std::cout<<"\t\t ULAClass::nfdc_beamforming | this->signalMatrix.shape (before slicing) = "<<
                    this->signalMatrix.sizes().vec() <<std::endl;
599             this->signalMatrix = \
600                 this->signalMatrix.index({torch::indexing::Slice(), \
601                                           torch::indexing::Slice(0, ideal_length), \
602                                           torch::indexing::Slice()});
603             std::cout<<"\t\t ULAClass::nfdc_beamforming | this->signalMatrix.shape (after slicing) = "<<
                    this->signalMatrix.sizes().vec() <<std::endl;
604             PRINTSPACE
605
606         }
607         else {
608             // creating a zero-filled tensor to append to signal matrix
609             torch::Tensor zero_tensor = torch::zeros({this->signalMatrix.size(0),          \
610                                                       num_zeros_to_pad_signal_along_dimension_0, \
611                                                       this->num_sensors}).to(torch::kFloat);
612
613             // appending to signal matrix
614             this->signalMatrix       = torch::cat({this->signalMatrix,                     \
615                                                    zero_tensor}, 1);
616         }
617
618         // breaking the signal into frames
619         fBuffer2D(this->signalMatrix, frame_size);
620
621         // add some zeros to the end of frames to accomodate delaying of signals.
622         torch::Tensor zero_filled_tensor =                \
623             torch::zeros({this->signalMatrix.size(0),     \
624                           this->num_buffer_zeros_per_frame, \
625                           this->num_sensors}).to(torch::kFloat);
626         this->signalMatrix =                              \
627             torch::cat({this->signalMatrix,               \
628                         zero_filled_tensor}, 1);
629
```

```
630         // tiling it to ensure that it works for all range-angle combinations
631         int number_of_azimuths_to_image = this->azimuth_centers.numel();
632         this->signalMatrix = \
633             torch::tile(this->signalMatrix, \
634                         {number_of_azimuths_to_image, 1, 1});
635 
636         // element-wise multiplying the signals to delay each of the frame accordingly
637         this->signalMatrix = torch::mul(this->signalMatrix, \
638                                         this->mulFFTMatrix);
639 
640         // summing up the signals
641         this->signalMatrix = torch::sum(this->signalMatrix, \
642                                         2,                  \
643                                         true);
644         // this->signalMatrix = torch::sum(this->signalMatrix, 2, false);
645 
646         // printing some stuff
647         std::cout<<"\t\t ULAClass::nfdc_beamforming: this->azimuth_centers.numel() =
                "<<this->azimuth_centers.numel() <<std::endl;
648         std::cout<<"\t\t ULAClass::nfdc_beamforming: this->range_centers.numel() =
                "<<this->range_centers.numel() <<std::endl;
649         std::cout<<"\t\t ULAClass::nfdc_beamforming: total number = "<<this->range_centers.numel() *
                this->azimuth_centers.numel() <<std::endl;
650         std::cout<<"\t\t ULAClass::nfdc_beamforming: this->signalMatrix.sizes().vec() =
                "<<this->signalMatrix.sizes().vec() <<std::endl;
651     }
652 
653     /* ============================================================================
654     Aim: create acoustic image directly
655     ---------------------------------------------------------------------------- */
656     void nfdc_createAcousticImage(ScattererClass* scatterers, \
657                                   TransmitterClass* transmitterObj){
658 
659         // first we ensure that the scattersers are in our frame of reference
660         scatterers->coordinates = scatterers->coordinates - this->location;
661 
662         // finding the spherical coordinates of the scatterers
663         torch::Tensor scatterers_spherical = fCart2Sph(scatterers->coordinates);
664 
665         // 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.
666         scatterers_spherical.index_put_({1, torch::indexing::Slice()}, 0);
667 
668         // converting the points back to cartesian
669         torch::Tensor scatterers_acoustic_cartesian = fSph2Cart(scatterers_spherical);
670 
671         // removing the z-dimension
672         scatterers_acoustic_cartesian = \
673             scatterers_acoustic_cartesian.index({torch::indexing::Slice(0, 2, 1), \
674                                                  torch::indexing::Slice()});
675 
676         // deciding image dimensions
677         int num_pixels_x = 512;
678         int num_pixels_y = 512;
679         torch::Tensor acousticImage =                   \
680             torch::zeros({num_pixels_x,                 \
681                           num_pixels_y}).to(torch::kFloat);
682 
683         // finding the max and min values
684         torch::Tensor min_x   = torch::min(scatterers_acoustic_cartesian[0]);
685         torch::Tensor max_x   = torch::max(scatterers_acoustic_cartesian[0]);
686         torch::Tensor min_y   = torch::min(scatterers_acoustic_cartesian[1]);
687         torch::Tensor max_y   = torch::max(scatterers_acoustic_cartesian[1]);
688 
689         // creating query grids
690         torch::Tensor query_x = torch::linspace(0, 1, num_pixels_x);
691         torch::Tensor query_y = torch::linspace(0, 1, num_pixels_y);
692 
693         // scaling it up to image max-point spread
694         query_x           = min_x + (max_x - min_x) * query_x;
695         query_y           = min_y + (max_y - min_y) * query_y;
```

```
696         float delta_queryx = (query_x[1] - query_x[0]).item<float>();
697         float delta_queryy = (query_y[1] - query_y[0]).item<float>();
698 
699         // creating a mesh-grid
700         auto queryMeshGrid = torch::meshgrid({query_x, query_y}, "ij");
701         query_x = queryMeshGrid[0].reshape({1, queryMeshGrid[0].numel()});
702         query_y = queryMeshGrid[1].reshape({1, queryMeshGrid[1].numel()});;
703         torch::Tensor queryMatrix = torch::cat({query_x, query_y}, 0);
704 
705         // printing shapes
706         if(DEBUG_ULA) PRINTSMALLLINE
707         if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: query_x.shape =
                "<<query_x.sizes().vec()<<std::endl;
708         if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: query_y.shape =
                "<<query_y.sizes().vec()<<std::endl;
709         if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: queryMatrix.shape =
                "<<queryMatrix.sizes().vec()<<std::endl;
710 
711         // setting up threshold values
712         float threshold_value =      \
713             std::min(delta_queryx,   \
714                     delta_queryy); if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line
                        711"<<std::endl;
715 
716         // putting a loop through the whole thing
717         for(int i = 0; i<queryMatrix[0].numel(); ++i){
718             // for each element in the query matrix
719             if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 716"<<std::endl;
720 
721             // calculating relative position of all the points
722             torch::Tensor relativeCoordinates = \
723                 scatterers_acoustic_cartesian - \
724                 queryMatrix.index({torch::indexing::Slice(), i}).reshape({2, 1}); if(DEBUG_ULA)
                        std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 720"<<std::endl;
725 
726             // calculating distances between all the points and the query point
727             torch::Tensor relativeDistances = \
728                 torch::linalg_norm(relativeCoordinates, \
729                                    1, 0, true, \
730                                    torch::kFloat);if(DEBUG_ULA) std::cout<<"\t\t\t
                                        ULAClass::nfdc_createAcousticImage: line 727"<<std::endl;
731             // finding points that are within the threshold
732             torch::Tensor conditionMeetingPoints = \
733                 relativeDistances.squeeze() <= threshold_value;if(DEBUG_ULA) std::cout<<"\t\t\t
                        ULAClass::nfdc_createAcousticImage: line 729"<<std::endl;
734 
735             // subsetting the points in the neighbourhood
736             if(torch::sum(conditionMeetingPoints).item<float>() == 0){
737 
738                 // continuing implementation if there are no points in the neighbourhood
739                 continue; if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line
                        735"<<std::endl;
740             }
741             else{
742                 // creating mask for points in the neighbourhood
743                 auto mask = (conditionMeetingPoints == 1);if(DEBUG_ULA) std::cout<<"\t\t\t
                        ULAClass::nfdc_createAcousticImage: line 739"<<std::endl;
744 
745                 // subsetting relative distances in the neighbourhood
746                 torch::Tensor distanceInTheNeighbourhood = \
747                     relativeDistances.index({torch::indexing::Slice(), mask});if(DEBUG_ULA) std::cout<<"\t\t\t
                            ULAClass::nfdc_createAcousticImage: line 743"<<std::endl;
748 
749                 // subsetting reflectivity of points in the neighbourhood
750                 torch::Tensor reflectivityInTheNeighbourhood = \
751                     scatterers->reflectivity.index({torch::indexing::Slice(), mask});if(DEBUG_ULA)
                            std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 747"<<std::endl;
752 
753                 // assigning intensity as a function of distance and reflectivity
754                 torch::Tensor reflectivityAssignment =                    \
755                     torch::mul(torch::exp(-distanceInTheNeighbourhood),   \
756                                reflectivityInTheNeighbourhood);if(DEBUG_ULA) std::cout<<"\t\t\t
                                    ULAClass::nfdc_createAcousticImage: line 752"<<std::endl;
```

```
757                 reflectivityAssignment = \
758                     torch::sum(reflectivityAssignment);if(DEBUG_ULA) std::cout<<"\t\t\t
                            ULAClass::nfdc_createAcousticImage: line 754"<<std::endl;
759
760                 // assigning this value to the image pixel intensity
761                 int pixel_position_x = i%num_pixels_x;
762                 int pixel_position_y = std::floor(i/num_pixels_x);
763                 acousticImage.index_put_({pixel_position_x, \
764                                           pixel_position_y}, \
765                                          reflectivityAssignment.item<float>());if(DEBUG_ULA) std::cout<<"\t\t\t
                                              ULAClass::nfdc_createAcousticImage: line 761"<<std::endl;
766             }
767
768         }
769
770         // storing the acoustic-image to the member
771         this->currentArtificalAcousticImage = acousticImage;
772
773         // // saving the torch::tensor
774         // torch::save(acousticImage, \
775         //             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/Assets/acoustic_image.pt");
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792         // // bringing it back to the original coordinates
793         // scatterers->coordinates = scatterers->coordinates + this->location;
794     }
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816 };
```

8.1.4 Class: Autonomous Underwater Vehicle

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

```
#include "ScattererClass.h"
#include "TransmitterClass.h"
#include "ULAClass.h"
#include <iostream>
#include <ostream>
#include <torch/torch.h>
#include <cmath>


// including functions
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fGetCurrentTimeFormatted.cpp"
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fCart2Sph.cpp"

#pragma once

// function to plot the thing
void fPlotTensors(){
    system("python /Users/vrsreeganesh/Documents/GitHub/AUV/Code/Python/TestingSaved_tensors.py");
}


void fSaveSeafloorScatteres(ScattererClass scatterer, \
                            ScattererClass scatterer_fls, \
                            ScattererClass scatterer_port, \
                            ScattererClass scatterer_starboard){

    // saving the ground-truth
    ScattererClass SeafloorScatter_gt = scatterer;
    torch::save(SeafloorScatter_gt.coordinates,
         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt.pt");
    torch::save(SeafloorScatter_gt.reflectivity,
         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt_reflectivity.pt");

    // saving coordinates
    torch::save(scatterer_fls.coordinates,
         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_fls_coordinates.pt");
    torch::save(scatterer_port.coordinates,
         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_port_coordinates.pt");
    torch::save(scatterer_starboard.coordinates,
         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_starboard.coordinates.pt");

    // saving reflectivities
    torch::save(scatterer_fls.reflectivity,
         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_fls_coordinates_reflectivity.pt");
    torch::save(scatterer_port.reflectivity,
         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_port_coordinates_reflectivity.pt");
    torch::save(scatterer_starboard.reflectivity,
         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_starboard.coordinates_reflectivity.pt");

    // plotting tensors
    fPlotTensors();

    // // saving the tensors
    // if (true) {

    //     // getting time ID
    //     auto timeID = fGetCurrentTimeFormatted();

    //     std::cout<<"\t\t\t\t\t\t\t Saving Tensors (timeID: "<<timeID<<")"<<std::endl;

    //     // saving the ground-truth
    //     ScattererClass SeafloorScatter_gt = scatterer;
    //     torch::save(SeafloorScatter_gt.coordinates, \
    //                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt.pt");
    //     torch::save(SeafloorScatter_gt.reflectivity, \
    //
         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt_reflectivity.pt");
```

```
59
60
61      //    // saving coordinates
62      //    torch::save(scatterer_fls.coordinates, \
63      //
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_fls_coordinates.pt");
64      //    torch::save(scatterer_port.coordinates, \
65      //
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_port_coordinates.pt");
66      //    torch::save(scatterer_starboard.coordinates, \
67      //
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_starboard.coordinates.pt");
68
69      //    // saving reflectivities
70      //    torch::save(scatterer_fls.reflectivity, \
71      //
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_fls_coordinates_reflectivity.pt");
72      //    torch::save(scatterer_port.reflectivity, \
73      //
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_port_coordinates_reflectivity.pt");
74      //    torch::save(scatterer_starboard.reflectivity, \
75      //
            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_starboard.coordinates_reflectivity.pt");
76
77      //    // plotting tensors
78      //    fPlotTensors();
79
80      //    // indicating end of thread
81      //    std::cout<<"\t\t\t\t\t\t\t Ended (timeID: "<<timeID<<")"<<std::endl;
82      // }
83  }
84
85  // including class-definitions
86  #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/ScattererClass.h"
87
88  // hash defines
89  #ifndef PRINTSPACE
90  #define PRINTSPACE    std::cout<<"\n\n\n\n\n\n\n\n"<<std::endl;
91  #endif
92  #ifndef PRINTSMALLLINE
93  #define PRINTSMALLLINE std::cout<<"-----------------------------------------------"<<std::endl;
94  #endif
95  #ifndef PRINTLINE
96  #define PRINTLINE     std::cout<<"==============================================="<<std::endl;
97  #endif
98
99  #ifndef DEVICE
100 #define DEVICE        torch::kMPS
101 // #define DEVICE        torch::kCPU
102 #endif
103
104 #define PI            3.14159265
105 // #define DEBUGMODE_AUV true
106 #define DEBUGMODE_AUV false
107
108
109 class AUVClass{
110 public:
111     // Intrinsic attributes
112     torch::Tensor location;          // location of vessel
113     torch::Tensor velocity;          // current speed of the vessel [a vector]
114     torch::Tensor acceleration;      // current acceleration of vessel [a vector]
115     torch::Tensor pointing_direction; // direction to which the AUV is pointed
116
117     // uniform linear-arrays
118     ULAClass ULA_fls;                // front-looking SONAR ULA
119     ULAClass ULA_port;               // mounted ULA [object of class, ULAClass]
120     ULAClass ULA_starboard;          // mounted ULA [object of class, ULAClass]
121
122     // transmitters
123     TransmitterClass transmitter_fls;   // transmitter for front-looking SONAR
124     TransmitterClass transmitter_port;  // mounted transmitter [obj of class, TransmitterClass]
125     TransmitterClass transmitter_starboard; // mounted transmitter [obj of class, TransmitterClass]
```

```
126
127     // derived or dependent attributes
128     torch::Tensor signalMatrix_1;         // matrix containing the signals obtained from ULA_1
129     torch::Tensor largeSignalMatrix_1;    // matrix holding signal of synthetic aperture
130     torch::Tensor beamformedLargeSignalMatrix;// each column is the beamformed signal at each stop-hop
131
132     // plotting mode
133     bool plottingmode;  // to suppress plotting associated with classes
134
135     // spotlight mode related
136     torch::Tensor absolute_coords_patch_cart; // cartesian coordinates of patch
137
138     // Synthetic Aperture Related
139     torch::Tensor ApertureSensorLocations; // sensor locations of aperture
140
141
142     /*=========================================================================
143     Aim: stepping motion
144     -------------------------------------------------------------------------*/
145     void step(float timestep){
146
147         // updating location
148         this->location = this->location + this->velocity * timestep;
149         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 81 \n";
150
151         // updating attributes of members
152         this->syncComponentAttributes();
153         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 85 \n";
154     }
155
156
157
158     /*=========================================================================
159     Aim: updateAttributes
160     -------------------------------------------------------------------------*/
161     void syncComponentAttributes(){
162
163         // updating ULA attributes
164         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 97 \n";
165
166         // updating locations
167         this->ULA_fls.location       = this->location;
168         this->ULA_port.location      = this->location;
169         this->ULA_starboard.location = this->location;
170
171         // updating the pointing direction of the ULAs
172         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 99 \n";
173         torch::Tensor ula_fls_sensor_direction_spherical = fCart2Sph(this->pointing_direction);       //
                spherical coords
174         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 101 \n";
175         ula_fls_sensor_direction_spherical[0]          = ula_fls_sensor_direction_spherical[0] - 90;
176         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 98 \n";
177         torch::Tensor ula_fls_sensor_direction_cart    = fSph2Cart(ula_fls_sensor_direction_spherical);
178         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 100 \n";
179
180         this->ULA_fls.sensorDirection       = ula_fls_sensor_direction_cart; // assigning sensor directionf or
                ULA-FLS
181         this->ULA_port.sensorDirection      = -this->pointing_direction;     // assigning sensor direction for
                ULA-Port
182         this->ULA_starboard.sensorDirection = -this->pointing_direction;     // assigning sensor direction for
                ULA-Starboard
183         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 105 \n";
184
185         // // calling the function to update the arguments
186         // this->ULA_fls.buildCoordinatesBasedOnLocation();  if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line
                109 \n";
187         // this->ULA_port.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line
                111 \n";
188         // this->ULA_starboard.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) std::cout<<"\t AUVClass:
                line 113 \n";
189
190         // updating transmitter locations
191         this->transmitter_fls.location    = this->location;
```

```
192         this->transmitter_port.location  = this->location;
193         this->transmitter_starboard.location = this->location;
194         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 102 \n";
195 
196         // updating transmitter pointing directions
197         this->transmitter_fls.updatePointingAngle(     this->pointing_direction);
198         this->transmitter_port.updatePointingAngle(    this->pointing_direction);
199         this->transmitter_starboard.updatePointingAngle( this->pointing_direction);
200         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 108 \n";
201     }
202 
203 
204 
205     /*=========================================================================
206     Aim: operator overriding for printing
207     -------------------------------------------------------------------------*/
208     friend std::ostream& operator<<(std::ostream& os, AUVClass &auv){
209         os<<"\t location = "<<torch::transpose(auv.location, 0, 1)<<std::endl;
210         os<<"\t velocity = "<<torch::transpose(auv.velocity, 0, 1)<<std::endl;
211         return os;
212     }
213 
214 
215     /*=========================================================================
216     Aim: Subsetting Scatterers
217     -------------------------------------------------------------------------*/
218     void subsetScatterers(ScattererClass* scatterers,\
219                           TransmitterClass* transmitterObj,\
220                           float tilt_angle){
221 
222         // ensuring components are synced
223         this->syncComponentAttributes();
224         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 120 \n";
225 
226         // calling the method associated with the transmitter
227         if(DEBUGMODE_AUV) {std::cout<<"\t\t scatterers.shape = "; fPrintTensorSize(scatterers->coordinates);}
228         if(DEBUGMODE_AUV) std::cout<<"\t\t tilt_angle = "<<tilt_angle<<std::endl;
229         transmitterObj->subsetScatterers(scatterers, tilt_angle);
230         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 124 \n";
231     }
232 
233     // yaw-correction matrix
234     torch::Tensor createYawCorrectionMatrix(torch::Tensor pointing_direction_spherical, \
235                                             float target_azimuth_deg){
236 
237         // building parameters
238         torch::Tensor azimuth_correction       =
                 torch::tensor({target_azimuth_deg}).to(torch::kFloat).to(DEVICE) - \
239                                                  pointing_direction_spherical[0];
240         torch::Tensor azimuth_correction_radians = azimuth_correction * PI / 180;
241 
242         torch::Tensor yawCorrectionMatrix = \
243             torch::tensor({torch::cos(azimuth_correction_radians).item<float>(), \
244                            torch::cos(torch::tensor({90}).to(torch::kFloat).to(DEVICE)*PI/180 +
                                 azimuth_correction_radians).item<float>(), \
245                            (float)0,                                                  \
246                            torch::sin(azimuth_correction_radians).item<float>(), \
247                            torch::sin(torch::tensor({90}).to(torch::kFloat).to(DEVICE)*PI/180 +
                                 azimuth_correction_radians).item<float>(), \
248                            (float)0,                                                  \
249                            (float)0,                                                  \
250                            (float)0,                                                  \
251                            (float)1}).reshape({3,3}).to(torch::kFloat).to(DEVICE);
252 
253         // returning the matrix
254         return yawCorrectionMatrix;
255     }
256 
257     // pitch-correction matrix
258     torch::Tensor createPitchCorrectionMatrix(torch::Tensor pointing_direction_spherical, \
259                                               float target_elevation_deg){
260 
261         // building parameters
```

```
262         torch::Tensor elevation_correction       =
                torch::tensor({target_elevation_deg}).to(torch::kFloat).to(DEVICE) - \
263                                                  pointing_direction_spherical[1];
264         torch::Tensor elevation_correction_radians = elevation_correction * PI / 180;
265
266         // creating the matrix
267         torch::Tensor pitchCorrectionMatrix = \
268             torch::tensor({{(float)1,                                                  \
269                             (float)0,                                                  \
270                             (float)0,                                                  \
271                             (float)0,                                                  \
272                             torch::cos(elevation_correction_radians).item<float>(), \
273                             torch::cos(torch::tensor({90}).to(torch::kFloat).to(DEVICE)*PI/180 +
                                    elevation_correction_radians).item<float>(),\
274                             (float)0,                                                  \
275                             torch::sin(elevation_correction_radians).item<float>(), \
276                             torch::sin(torch::tensor({90}).to(torch::kFloat).to(DEVICE)*PI/180 +
                                    elevation_correction_radians).item<float>()}).reshape({3,3}).to(torch::kFloat);
277
278         // returning the matrix
279         return pitchCorrectionMatrix;
280     }
281
282     // Signal Simulation
283     void simulateSignal(ScattererClass& scatterer){
284
285         // making three copies
286         ScattererClass scatterer_fls     = scatterer;
287         ScattererClass scatterer_port    = scatterer;
288         ScattererClass scatterer_starboard = scatterer;
289
290         // printing size of scatterers before subsetting
291         std::cout<< "> AUVClass: Beginning Scatterer Subsetting"<<std::endl;
292         std::cout<<"\t AUVClass: scatterer_fls.coordinates.shape (before) = ";
                fPrintTensorSize(scatterer_fls.coordinates);
293         std::cout<<"\t AUVClass: scatterer_port.coordinates.shape (before) = ";
                fPrintTensorSize(scatterer_port.coordinates);
294         std::cout<<"\t AUVClass: scatterer_starboard.coordinates.shape (before) = ";
                fPrintTensorSize(scatterer_starboard.coordinates);
295
296         // finding the pointing direction in spherical
297         torch::Tensor auv_pointing_direction_spherical = fCart2Sph(this->pointing_direction);
298
299         // asking the transmitters to subset the scatterers by multithreading
300         std::thread transmitterFLSSubset_t(&AUVClass::subsetScatterers, this, \
301                                            &scatterer_fls,\
302                                            &this->transmitter_fls, \
303                                            (float)0);
304         std::thread transmitterPortSubset_t(&AUVClass::subsetScatterers, this, \
305                                             &scatterer_port,\
306                                             &this->transmitter_port, \
307                                             auv_pointing_direction_spherical[1].item<float>());
308         std::thread transmitterStarboardSubset_t(&AUVClass::subsetScatterers, this, \
309                                                  &scatterer_starboard, \
310                                                  &this->transmitter_starboard, \
311                                                  - auv_pointing_direction_spherical[1].item<float>());
312
313         // joining the subset threads back
314         transmitterFLSSubset_t.join(); transmitterPortSubset_t.join(); transmitterStarboardSubset_t.join();
315
316         // printing the size of these points before subsetting
317         PRINTDOTS
318         std::cout<<"\t AUVClass: scatterer_fls.coordinates.shape (after) = ";
                fPrintTensorSize(scatterer_fls.coordinates);
319         std::cout<<"\t AUVClass: scatterer_port.coordinates.shape (after) = ";
                fPrintTensorSize(scatterer_port.coordinates);
320         std::cout<<"\t AUVClass: scatterer_starboard.coordinates.shape (after) = ";
                fPrintTensorSize(scatterer_starboard.coordinates);
321
322         // multithreading the saving tensors part.
323         std::thread savetensor_t(fSaveSeafloorScatteres, \
324                                  scatterer,             \
325                                  scatterer_fls,         \
```

```
                            scatterer_port,         \
                            scatterer_starboard);

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

        // saving the tensors
        if (true) {
            // saving the ground-truth
            torch::save(this->ULA_fls.signalMatrix,
                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_fls.pt");
            torch::save(this->ULA_port.signalMatrix,
                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_port.pt");
            torch::save(this->ULA_starboard.signalMatrix,
                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_starboard.pt");
        }


    }

    // Imaging Function
    void image(){

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

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


        // performing the beamforming
        // std::thread ULA_fls_beamforming_t(&ULAClass::nfdc_beamforming,   \
        //                                  &this->ULA_fls,                 \
        //                                  &this->transmitter_fls);
        // std::thread ULA_port_beamforming_t(&ULAClass::nfdc_beamforming,  \
```

```
395         //                              &this->ULA_port,                \
396         //                              &this->transmitter_port);
397         // std::thread ULA_starboard_beamforming_t(&ULAClass::nfdc_beamforming, \
398         //                                  &this->ULA_starboard,       \
399         //                                  &this->transmitter_starboard);
400
401         // joining the filters back
402         // ULA_fls_beamforming_t.join();
403         // ULA_port_beamforming_t.join();
404         // ULA_starboard_beamforming_t.join();
405
406     }
407
408
409     /* ============================================================================
410     Aim: Init
411     ----------------------------------------------------------------------------*/
412     void init(){
413
414         // call sync-component attributes
415         this->syncComponentAttributes();
416
417         // initializing all the ULAs
418         this->ULA_fls.init(      &this->transmitter_fls);
419         this->ULA_port.init(     &this->transmitter_port);
420         this->ULA_starboard.init( &this->transmitter_starboard);
421
422         // precomputing delay-matrices for the ULA-class
423         std::thread ULA_fls_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
424                                             &this->ULA_fls,                 \
425                                             &this->transmitter_fls);
426         std::thread ULA_port_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
427                                              &this->ULA_port,               \
428                                              &this->transmitter_port);
429         std::thread ULA_starboard_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
430                                                  &this->ULA_starboard,      \
431                                                  &this->transmitter_starboard);
432
433         // joining the threads back
434         ULA_fls_precompute_weights_t.join();
435         ULA_port_precompute_weights_t.join();
436         ULA_starboard_precompute_weights_t.join();
437
438     }
439
440     /* ============================================================================
441     Aim: directly create acoustic image
442     ---------------------------------------------------------------------------- */
443     void createAcousticImage(ScattererClass* scatterers){
444
445         // making three copies
446         ScattererClass scatterer_fls     = scatterers;
447         ScattererClass scatterer_port    = scatterers;
448         ScattererClass scatterer_starboard = scatterers;
449
450         // printing size of scatterers before subsetting
451         PRINTSMALLLINE
452         std::cout<< "\t > AUVClass::createAcousticImage: Beginning Scatterer Subsetting"<<std::endl;
453         std::cout<<"\t AUVClass::createAcousticImage: scatterer_fls.coordinates.shape (before) = ";
                fPrintTensorSize(scatterer_fls.coordinates);
454         std::cout<<"\t AUVClass::createAcousticImage: scatterer_port.coordinates.shape (before) = ";
                fPrintTensorSize(scatterer_port.coordinates);
455         std::cout<<"\t AUVClass::createAcousticImage: scatterer_starboard.coordinates.shape (before) = ";
                fPrintTensorSize(scatterer_starboard.coordinates);
456
457         // finding the pointing direction in spherical
458         torch::Tensor auv_pointing_direction_spherical = fCart2Sph(this->pointing_direction);
459
460         // asking the transmitters to subset the scatterers by multithreading
461         std::thread transmitterFLSSubset_t(&AUVClass::subsetScatterers, this, \
462                                       &scatterer_fls,\
463                                       &this->transmitter_fls, \
464                                       (float)0);
```

```
465         std::thread transmitterPortSubset_t(&AUVClass::subsetScatterers, this, \
466                                         &scatterer_port,\
467                                         &this->transmitter_port, \
468                                         auv_pointing_direction_spherical[1].item<float>());
469         std::thread transmitterStarboardSubset_t(&AUVClass::subsetScatterers, this, \
470                                             &scatterer_starboard, \
471                                             &this->transmitter_starboard, \
472                                             - auv_pointing_direction_spherical[1].item<float>());
473 
474         // joining the subset threads back
475         transmitterFLSSubset_t.join(    );
476         transmitterPortSubset_t.join(   );
477         transmitterStarboardSubset_t.join( );
478 
479 
480         // asking the ULAs to directly create acoustic images
481         std::thread ULA_fls_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, this->ULA_fls, \
482                                         &scatterer_fls, &this->transmitter_fls);
483         std::thread ULA_port_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, &this->ULA_port, \
484                                         &scatterer_port, &this->transmitter_port);
485         std::thread ULA_starboard_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, &this->ULA_starboard, \
486                                             &scatterer_starboard, &this->transmitter_starboard);
487 
488         // joining the threads back
489         ULA_fls_acoustic_image_t.join(  );
490         ULA_port_acoustic_image_t.join( );
491         ULA_starboard_acoustic_image_t.join();
492 
493     }
494 
495 
496 };
```

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

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

// 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
    torch::Tensor box_coordinates = torch::zeros({3,1}).to(torch::kFloat).to(DEVICE);
    torch::Tensor box_reflectivity = torch::zeros({1,1}).to(torch::kFloat).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 = torch::linspace(0, \
                                bed_width, \
                                bed_width * bed_width_density).to(DEVICE);
    auto ypoints = torch::linspace(0, \
                                bed_length, \
                                bed_length * bed_length_density).to(DEVICE);

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

```

```
66      // creating heights of scattereres
67      if(HILLS == true){
68
69          // setting up hill parameters
70          int num_hills = 10;
71
72          // setting up placement of hills
73          torch::Tensor points2D = torch::cat({X, Y}, 0);
74          torch::Tensor min2D   = std::get<0>(torch::min(points2D, 1, true));
75          torch::Tensor max2D   = std::get<0>(torch::max(points2D, 1, true));
76          torch::Tensor hill_means = \
77              min2D                                          \
78              + torch::mul(torch::rand({2, num_hills}), \
79                          max2D - min2D);
80
81          // setting up hill dimensions
82          torch::Tensor hill_dimensions_min = \
83              torch::tensor({10, \
84                             10, \
85                             2}).reshape({3,1});
86          torch::Tensor hill_dimensions_max = \
87              torch::tensor({30, \
88                             30, \
89                             7}).reshape({3,1});
90          torch::Tensor hill_dimensions = \
91              hill_dimensions_min + \
92              torch::mul(hill_dimensions_max - hill_dimensions_min, \
93                         torch::rand({3, num_hills}));
94
95          // calling the hill-creation function
96          fCreateHills(hill_means, \
97                       hill_dimensions, \
98                       points2D);
99
100         // setting up floor reflectivity
101         torch::Tensor floorScatter_reflectivity = \
102             torch::ones({1, Y.numel()}).to(DEVICE);
103
104         // populating the values of the incoming argument.
105         scatterers->coordinates  = points2D; // assigning coordinates
106         scatterers->reflectivity = floorScatter_reflectivity;// assigning reflectivity
107     }
108     else{
109
110         // assigning flat heights
111         torch::Tensor Z = torch::zeros({1, Y.numel()}).to(DEVICE);
112
113         // setting up floor coordinates
114         torch::Tensor floorScatter_coordinates = torch::cat({X, Y, Z}, 0);
115         torch::Tensor floorScatter_reflectivity = torch::ones({1, Y.numel()}).to(DEVICE);
116
117         // populating the values of the incoming argument.
118         scatterers->coordinates  = floorScatter_coordinates; // assigning coordinates
119         scatterers->reflectivity = floorScatter_reflectivity;// assigning reflectivity
120     }
121
122     // combining the values
123     if(DEBUG_SEAFLOOR) std::cout<<"\t SeafloorSetup: line 166 \n";
124     if(DEBUG_SEAFLOOR) {std::cout<<"\t scatterers->coordinates.shape = ";
            fPrintTensorSize(scatterers->coordinates);}
125     if(DEBUG_SEAFLOOR) {std::cout<<"\t box_coordinates.shape = "; fPrintTensorSize(box_coordinates);}
126     if(DEBUG_SEAFLOOR) {std::cout<<"\t scatterers->reflectivity.shape = ";
            fPrintTensorSize(scatterers->reflectivity);}
127     if(DEBUG_SEAFLOOR) {std::cout<<"\t box_reflectivity = "; fPrintTensorSize(box_reflectivity);}
128
129
130     // assigning values to the coordinates
131     scatterers->coordinates  = torch::cat({scatterers->coordinates, box_coordinates}, 1);
132     scatterers->reflectivity = torch::cat({scatterers->reflectivity, box_reflectivity}, 1);
133
134     // saving tensors
135     if(SAVETENSORS_Seafloor){
136         torch::save(scatterers->coordinates, \
```

```
137                     "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt.pt");
138         std::cout<<"SeafloorSetup: Saved Seafloor "<<std::endl;
139     }
140
141 }
```

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        = 0.2;                   // 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
```

```
67      transmitter_fls->elevation_angle       = elevation_angle;     // assigning elevation angle
68      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       torch::kMPS
    #define DEVICE       torch::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   = 0.25;                        // sampling-period

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

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

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

    // 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
```

```
59      ula_port->lowpassFilterCoefficientsForDecimation = lowpassfiltercoefficients;
60
61
62      // assigning values
63      ula_starboard->num_sensors        = num_sensors;           // assigning number of sensors
64      ula_starboard->inter_element_spacing = inter_element_spacing; // assigning inter-element spacing
65      ula_starboard->coordinates        = ULA_coordinates;       // assigning ULA coordinates
66      ula_starboard->sampling_frequency = sampling_frequency;    // assigning sampling frequencys
67      ula_starboard->recording_period   = recording_period;      // assigning recording period
68      ula_starboard->sensorDirection    = ULA_direction;         // ULA direction
69      ula_starboard->lowpassFilterCoefficientsForDecimation = lowpassfiltercoefficients;
70
71
72  }
```

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(torch::kFloat).to(DEVICE); // starting location of AUV
    torch::Tensor velocity        = torch::tensor({5,0, 0}).reshape({3,1}).to(torch::kFloat).to(DEVICE); // starting velocity of AUV
    torch::Tensor pointing_direction = torch::tensor({1,0, 0}).reshape({3,1}).to(torch::kFloat).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, torch::kFloat).to(DEVICE);
    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, torch::kFloat).to(DEVICE);
    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(torch::kFloat).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;

    // 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;

    // over-riding the result with the input so that we can save memory
    inputMatrix = convolvedResult; 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){

    // ensuring the first dimension is 1.
    if(inputMatrix.size(0) != 1){
        throw std::runtime_error("fBuffer2D: The first-dimension must be 1 \n");
    }
```

```

    // padding with zeros in case it is not a perfect multiple
    if(inputMatrix.size(1)%frame_size != 0){
        // padding with zeros
        int numberofzeroestoadd = frame_size - (inputMatrix.size(1) % frame_size);
        if(DEBUG_Buffer2D) {
            std::cout  << "\t\t\t fBuffer2D: frame_size = "               << frame_size             <<
                 std::endl;
            std::cout  << "\t\t\t fBuffer2D: inputMatrix.sizes().vec() = " << inputMatrix.sizes().vec() <<
                 std::endl;
            std::cout  << "\t\t\t fBuffer2D: numberofzeroestoadd = "    << numberofzeroestoadd    << std::endl;
        }

        // creating zero matrix
        torch::Tensor zeroMatrix = torch::zeros({inputMatrix.size(0), \
                                            numberofzeroestoadd, \
                                            inputMatrix.size(2)});
        if(DEBUG_Buffer2D) std::cout<<"\t\t\t fBuffer2D: zeroMatrix.sizes() =
             "<<zeroMatrix.sizes().vec()<<std::endl;

        // adding the zero matrix
        inputMatrix = torch::cat({inputMatrix, zeroMatrix}, 1);
        if(DEBUG_Buffer2D) std::cout<<"\t\t\t fBuffer2D: inputMatrix.sizes().vec() =
             "<<inputMatrix.sizes().vec()<<std::endl;
    }

    // calculating some parameters
    // int num_frames = inputMatrix.size(1)/frame_size;
    int num_frames = std::ceil(inputMatrix.size(1)/frame_size);
    if(DEBUG_Buffer2D) std::cout << "\t\t\t fBuffer2D: inputMatrix.sizes = "<< inputMatrix.sizes().vec()<<
         std::endl;
    if(DEBUG_Buffer2D) std::cout << "\t\t\t fBuffer2D: framesize = " << frame_size            << std::endl;
    if(DEBUG_Buffer2D) std::cout << "\t\t\t fBuffer2D: num_frames = " << num_frames          << std::endl;

    // defining target shape and size
    std::vector<int64_t> target_shape = {num_frames,                    \
                                         frame_size,                    \
                                         inputMatrix.size(2)};
    std::vector<int64_t> target_strides = {frame_size * inputMatrix.size(2), \
                                         inputMatrix.size(2),           \
                                         1};
    if(DEBUG_Buffer2D) std::cout << "\t\t\t fBuffer2D: STATUS: created shape and strides"<< std::endl;

    // creating the transformation
    inputMatrix = inputMatrix.as_strided(target_shape, target_strides);

}
```

8.3.5 fAnglesToTensor

```
#include <torch/torch.h>
// function: angles to vector
torch::Tensor fAnglesToTensor(float azimuthal_angle,
                              float elevation_angle)
{
  // calculating tensor
  torch::Tensor coordinateTensor = torch::tensor({cos(elevation_angle) * cos(azimuthal_angle),
                                          cos(elevation_angle) * sin(azimuthal_angle),
                                          sin(elevation_angle)}).view({3,1});

  // returning value
  return coordinateTensor;
}
```

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 standard
#include <cstdint>
#include <ostream>
#include <torch/torch.h>
#include <iostream>
#include <thread>
#include "math.h"
#include <chrono>
#include <Python.h>
#include <cstdlib>


// hash defines
#ifndef PRINTSPACE
#define PRINTSPACE    std::cout<<"\n\n\n";
#endif
#ifndef PRINTSMALLLINE
#define PRINTSMALLLINE
     std::cout<<"----------------------------------------------------------------------------------------"<<std::endl;
#endif
#ifndef PRINTDOTS
#define PRINTDOTS
     std::cout<<"........................................................................."<<std::endl;
#endif
#ifndef PRINTLINE
#define PRINTLINE
     std::cout<<"========================================================================================"<<std::endl;
#endif
#ifndef PI
#define PI            3.14159265
#endif

// debugging hashdefine
#ifndef DEBUGMODE
#define DEBUGMODE     false
#endif

// deciding to save tensors or not
#ifndef SAVETENSORS
    #define SAVETENSORS     true
    // #define SAVETENSORS    false
#endif

// choose device here
#ifndef DEVICE
    #define DEVICE        torch::kCPU
    // #define DEVICE        torch::kMPS
    // #define DEVICE        torch::kCUDA
#endif

// class definitions
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/ScattererClass.h"
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/ULAClass.h"
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/TransmitterClass.h"
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/AUVClass.h"

// setup-scripts
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/ULASetup/ULASetup.cpp"
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/TransmitterSetup/TransmitterSetup.cpp"
```

```
62 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/SeafloorSetup/SeafloorSetup.cpp"
63 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/AUVSetup/AUVSetup.cpp"
64
65 // functions
66 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fPrintTensorSize.cpp"
67 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fSph2Cart.cpp"
68 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fCart2Sph.cpp"
69 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fConvolveColumns.cpp"
70
71
72 // main-function
73 int main() {
74
75     // Builing Sea-floor
76     ScattererClass SeafloorScatter;
77     std::thread scatterThread_t(SeafloorSetup, \
78                                 &SeafloorScatter);
79
80     // Building ULA
81     ULAClass ula_fls, ula_port, ula_starboard;
82     std::thread ulaThread_t(ULASetup, \
83                             &ula_fls, \
84                             &ula_port, \
85                             &ula_starboard);
86
87     // Building Transmitter
88     TransmitterClass transmitter_fls, transmitter_port, transmitter_starboard;
89     std::thread transmitterThread_t(TransmitterSetup,
90                                     &transmitter_fls,
91                                     &transmitter_port,
92                                     &transmitter_starboard);
93
94     // Joining threads
95     ulaThread_t.join();       // making the ULA population thread join back
96     transmitterThread_t.join(); // making the transmitter population thread join back
97     scatterThread_t.join();  // making the scattetr population thread join back
98
99     // building AUV
100    AUVClass auv;             // instantiating class object
101    AUVSetup(&auv);      // populating
102
103    // attaching components to the AUV
104    auv.ULA_fls              = ula_fls;              // attaching ULA-FLS to AUV
105    auv.ULA_port             = ula_port;             // attaching ULA-Port to AUV
106    auv.ULA_starboard        = ula_starboard;        // attaching ULA-Starboard to AUV
107    auv.transmitter_fls      = transmitter_fls;      // attaching Transmitter-FLS to AUV
108    auv.transmitter_port     = transmitter_port;     // attaching Transmitter-Port to AUV
109    auv.transmitter_starboard = transmitter_starboard; // attaching Transmitter-Starboard to AUV
110
111    // storing
112    ScattererClass SeafloorScatter_deepcopy = SeafloorScatter;
113
114    // pre-computing the imaging matrices
115    auv.init();
116
117    // mimicking movement
118    int number_of_stophops = 1;
119    if (true) return 0;
120    for(int i = 0; i<number_of_stophops; ++i){
121
122        // time measuring
123        auto start_time = std::chrono::high_resolution_clock::now();
124
125        // printing some spaces
126        PRINTSPACE; PRINTSPACE; PRINTLINE; std::cout<<"i = "<<i<<std::endl; PRINTLINE
127
128        // making the deep copy
129        ScattererClass SeafloorScatter  = SeafloorScatter_deepcopy; // copy for FLS
130
131        // simulating the signals received in this time step
132        auv.simulateSignal(SeafloorScatter);
133
134        // decimating the signal received in this time step
```

```
135         auv.image();
136 
137         // measuring time
138         auto end_time  = std::chrono::high_resolution_clock::now();
139         std::chrono::duration<double> time_duration = end_time - start_time;
140         PRINTDOTS; std::cout<<"Time taken (i = "<<i<<") = "<<time_duration.count()<<" seconds"<<std::endl;
                PRINTDOTS
141 
142         // moving to next position
143         auv.step(0.5);
144 
145     }
146 
147 
148 
149 
150 
151 
152 
153 
154 
155 
156 
157 
158 
159 
160 
161 
162 
163 
164 
165 
166 
167 
168 
169 
170 
171 
172 
173 
174 
175 
176 
177 
178 
179 
180 
181 
182 
183 
184 
185 
186 
187 
188 
189 
190 
191 
192 
193 
194 
195 
196 
197 
198 
199 
200 
201 
202 
203 
204 
205     // // encapsulating coordinates and reflectivity in a dictionary
206     // std::unordered_map<std::string, torch::Tensor> floor_scatterers;
```

```
207 // torch::load(floor_scatterers["coordinates"],
208 //           "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/floor_coordinates_3D.pt");
209 // torch::load(floor_scatterers["reflectivity"],
210 //           "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/floor_scatterers_reflectivity.pt");
211
212 // // sending to GPU
213 // floor_scatterers["coordinates"] = floor_scatterers["coordinates"].to( torch::kMPS);
214 // floor_scatterers["reflectivity"] = floor_scatterers["reflectivity"].to( torch::kMPS);
215
216
217
218
219 // // AUV Setup
220 // torch::Tensor auv_initial_location       = torch::tensor({0.0, 2.0, 2.0}).view({3,1}).to(torch::kMPS);
        // initial location
221 // torch::Tensor auv_initial_velocity       = torch::tensor({1.0, 0.0, 0.0}).view({3,1}).to(torch::kMPS);
        // initial velocity
222 // torch::Tensor auv_initial_acceleration   = torch::tensor({0.0, 0.0, 0.0}).view({3,1}).to(torch::kMPS);
        // initial acceleration
223 // torch::Tensor auv_initial_pointing_direction = torch::tensor({1.0, 0.0,
        0.0}).view({3,1}).to(torch::kMPS); // initial pointing direction
224
225 // // Initializing a member of class, AUV
226 // AUV auv(auv_initial_location,          // assigning initial location
227 //        auv_initial_velocity,           // assigning initial velocity
228 //        auv_initial_acceleration,       // assigning initial acceleration
229 //        auv_initial_pointing_direction); // assigning initial pointing direction
230
231
232
233 // // Setting up ULAs for the AUV: front, portside and starboard
234 // const int num_sensors         = 32;                // number of sensors
235 // const double intersensor_distance = 1e-4;          // distance between sensors
236
237 // ULA ula_portside(num_sensors, intersensor_distance); // ULA onbject for portside
238 // ULA ula_fbls(num_sensors,    intersensor_distance); // ULA object for front-side
239 // ULA ula_starboard(num_sensors, intersensor_distance); // ULA object for starboard
240
241 // auv.ula_portside              = ula_portside;      // attaching portside-ULAs to the AUV
242 // auv.ula_fbls                  = ula_fbls;          // attaching front-ULA to the AUV
243 // auv.ula_starboard             = ula_starboard;     // attaching starboard-ULA to the AUV
244
245
246
247 // // Setting up Projector: front, portside and starboard
248 // Projector projector_portside(torch::zeros({3,1}).to(torch::kMPS), // location
249 //                           fDeg2Rad(90),                      // azimuthal angle
250 //                           fDeg2Rad(-30),                     // elevation angle
251 //                           fDeg2Rad(30),                      // azimuthal beamwidth
252 //                           fDeg2Rad(20));                     // elevation beamwidth
253 // Projector projector_fbls(torch::zeros({3,1}).to(torch::kMPS), // location
254 //                       fDeg2Rad(0),                           // azimuthal angle
255 //                       fDeg2Rad(-30),                         // elevation angle
256 //                       fDeg2Rad(120),                         // azimuthal beamwidth
257 //                       fDeg2Rad(60));                         // elevation beamwidth;
258 // Projector projector_starboard(torch::zeros({3,1}).to(torch::kMPS), // location
259 //                           fDeg2Rad(-90),                     // azimuthal angle
260 //                           fDeg2Rad(-30),                     // elevation angle
261 //                           fDeg2Rad(30),                      // azimuthal beamwidth
262 //                           fDeg2Rad(20));                     // elevation beamwidth;
263
264 // auv.projector_portside = projector_portside;        // Attaching projectors to AUV
265 // auv.projector_fbls    = projector_fbls;             // Attaching projectors to AUV
266 // auv.projector_starboard = projector_starboard;     // Attaching projectors to AUV
267
268
269
270
271
272 // // testing projection
273 // torch::Tensor coordinates = torch::tensor({ 1, 2, 3, 4,
274 //                                          0,  0,  0,  0,
275 //                                         -1, -1, -1, -1}).view({3,4}).to(torch::kFloat).to(torch::kMPS);
```

```
276   // torch::Tensor coordinates_normalized = fColumnNormalize(coordinates);
277   // torch::Tensor coordinates_projected = coordinates.clone();
278   // coordinates_projected[2] = torch::zeros({coordinates.size(1)});
279
280   // torch::Tensor innerproduct = coordinates * coordinates_projected;
281   // innerproduct = torch::sum(innerproduct, 0, true);
282
283
284
285   // PRINTLINE
286   // torch::Tensor xy = coordinates.clone();
287   // xy[2] = torch::zeros({xy.size(1)});
288   // std::cout<<"coordinates = \n"<<coordinates<<std::endl;
289   // PRINTSMALLLINE
290   // std::cout<<"xy = \n"<<xy<<std::endl;
291   // torch::Tensor xylengths = torch::norm(xy, 2, 0, true, torch::kFloat);
292   // std::cout<<"xylengths = \n"<<xylengths<<std::endl;
293   // PRINTLINE
294
295
296
297
298
299
300   // returning
301   return 0;
302 }
```

Chapter 9

Reading

9.1 Primary Books

1.

9.2 Interesting Papers