

Autonomous Underwater Vehicle: A Surveillance Protocol

S.V. Rajendran

February 16, 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

Preface	i
Introduction	ii
1 Setup	1
1.1 Overview	1
2 Underwater Environment Setup	2
2.1 Sea “Floor”	2
2.2 Simple Structures	3
2.2.1 Boxes	3
2.2.2 Sphere	4
3 Hardware Setup	5
3.1 Transmitter	5
3.2 Uniform Linear Array	6
3.3 Marine Vessel	6
4 Signal Simulation	7
4.1 Transmitted Signal	7
4.2 Signal Simulation	8
4.3 Ray Tracing	8
4.3.1 Pairwise Dot-Product	8
4.3.2 Range Histogram Method	9
5 Imaging	10
5.1 Decimation	10
5.1.1 Basebanding	10
5.1.2 Lowpass filtering	11
5.1.3 Decimation	11
5.2 Match-Filtering	11
6 Control Pipeline	14
7 Results	16
8 Software	17
8.1 Class Definitions	17
8.1.1 Class: Scatter	17

8.1.2	Class: Transmitter	19
8.1.3	Class: Uniform Linear Array	26
8.1.4	Class: Autonomous Underwater Vehicle	43
8.2	Setup Scripts	52
8.2.1	Seafloor Setup	52
8.2.2	Transmitter Setup	55
8.2.3	Uniform Linear Array	57
8.2.4	AUV Setup	59
8.3	Function Definitions	60
8.3.1	Cartesian Coordinates to Spherical Coordinates	60
8.3.2	Spherical Coordinates to Cartesian Coordinates	61
8.3.3	Column-Wise Convolution	61
8.3.4	Buffer 2D	62
8.3.5	fAnglesToTensor	63
8.3.6	fCalculateCosine	63
8.4	Main Scripts	65
8.4.1	Signal Simulation	65
9	Reading	69
9.1	Primary Books	69
9.2	Interesting Papers	69

Chapter 1

Setup

1.1 Overview

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

Chapter 2

Underwater Environment Setup

Overview

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

Sea-floors in real-life are rarely flat. They often contain valleys, mountains, hills and much richer geographical features. Thus, training an agent to function in such environments call for the creation of similar structures in our simulations. Even though there must be infinite variations in the structures found under water, we shall take a constrained and structured approach to creating these variations. To that end, we shall start with an additive approach. We define few types of underwater structure whos shape, size and what not can be parameterized to enable creation of random seafloors. The full-script for creating the sea-floor is available in section 8.2.1.

2.1 Sea “Floor”

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

The most basic approach to creating this is to create a flat seafloor, where all the points have the same height. While this is a good place to start, it is good to bring in some realism to the seafloor. To that end, we shall have some rolling hills as the sea-floor. Each “hill ” is created using the method outlined in Algorithm 1. The method involves deciding the location of the hills, the dimension of the hills and then designing a hill by combining an exponential function and a cosine function. We’re aiming to essentially produce gaussian-looking sea-floor hills. After the creation, this becomes the set of points representing the lowest set of points in the overall seafloor structure.

Algorithm 1 Hill Creation

```

1: Input: Mean vector  $\mathbf{m}$ , Dimension vector  $\mathbf{d}$ , 2D points  $\mathbf{P}$ 
2: Output: Updated  $\mathbf{P}$  with hill heights
3:  $\text{num\_hills} \leftarrow \text{numel}(\mathbf{m}_x)$ 
4:  $H \leftarrow$  Zeros tensor of size  $(1, \text{numel}(\mathbf{P}_x))$ 
5: for  $i = 1$  to  $\text{num\_hills}$  do
6:    $x_{\text{norm}} \leftarrow \frac{\frac{\pi}{2}(\mathbf{P}_x - \mathbf{m}_x[i])}{\mathbf{d}_x[i]}$ 
7:    $y_{\text{norm}} \leftarrow \frac{\frac{\pi}{2}(\mathbf{P}_y - \mathbf{m}_y[i])}{\mathbf{d}_y[i]}$ 
8:    $h_x \leftarrow \cos(x_{\text{norm}}) \cdot e^{\frac{|x_{\text{norm}}|}{10}}$ 
9:    $h_y \leftarrow \cos(y_{\text{norm}}) \cdot e^{\frac{|y_{\text{norm}}|}{10}}$ 
10:   $h \leftarrow \mathbf{d}_z[i] \cdot h_x \cdot h_y$ 
11:  Apply boundary conditions:
12:  if  $x_{\text{norm}} > \frac{\pi}{2}$  or  $x_{\text{norm}} < -\frac{\pi}{2}$  or  $y_{\text{norm}} > \frac{\pi}{2}$  or  $y_{\text{norm}} < -\frac{\pi}{2}$  then
13:     $h \leftarrow 0$ 
14:  end if
15:   $H \leftarrow H + h$ 
16: end for
17:  $\mathbf{P} \leftarrow \text{concatenate}([\mathbf{P}, H])$ 

```

2.2 Simple Structures

2.2.1 Boxes

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

Algorithm 2 Generate Box Meshes on Sea Floor

Require: *across_track_length*, *along_track_length*, *box_coordinates*, *box_reflectivity*

- 1: **Initialize** min/max width, length, height, meshdensity, reflectivity, and number of boxes
- 2: Generate random center points for boxes:
- 3: $midxypoints \leftarrow \text{rand}([3, num_boxes])$
- 4: $midxypoints[0] \leftarrow midxypoints[0] \times across_track_length$
- 5: $midxypoints[1] \leftarrow midxypoints[1] \times along_track_length$
- 6: $midxypoints[2] \leftarrow 0$
- 7: Assign random dimensions to each box:
- 8: $boxwidths \leftarrow \text{rand}(num_boxes) \times (max_width - min_width) + min_width$
- 9: $boxlengths \leftarrow \text{rand}(num_boxes) \times (max_length - min_length) + min_length$
- 10: $boxheights \leftarrow \text{rand}(num_boxes) \times (max_height - min_height) + min_height$
- 11: **for** $i = 1$ to num_boxes **do**
- 12: Generate mesh points along each axis:
- 13: $xpoints \leftarrow \text{linspace}(-boxwidths[i]/2, boxwidths[i]/2, boxwidths[i] \times meshdensity)$
- 14: $ypoints \leftarrow \text{linspace}(-boxlengths[i]/2, boxlengths[i]/2, boxlengths[i] \times meshdensity)$
- 15: $zpoints \leftarrow \text{linspace}(0, boxheights[i], boxheights[i] \times meshdensity)$
- 16: Generate 3D mesh grid:
- 17: $X, Y, Z \leftarrow \text{meshgrid}(xpoints, ypoints, zpoints)$
- 18: Reshape X, Y, Z into 1D tensors
- 19: Compute final coordinates:
- 20: $boxcoordinates \leftarrow \text{cat}(X, Y, Z)$
- 21: $boxcoordinates[0] \leftarrow boxcoordinates[0] + midxypoints[0][i]$
- 22: $boxcoordinates[1] \leftarrow boxcoordinates[1] + midxypoints[1][i]$
- 23: $boxcoordinates[2] \leftarrow boxcoordinates[2] + midxypoints[2][i]$
- 24: Generate reflectivity values:
- 25: $boxreflectivity \leftarrow meshreflectivity + \text{rand}(1, \text{size}(boxcoordinates)) - 0.5$
- 26: Append data to final tensors:
- 27: $box_coordinates \leftarrow \text{cat}(box_coordinates, boxcoordinates, 1)$
- 28: $box_reflectivity \leftarrow \text{cat}(box_reflectivity, boxreflectivity, 1)$
- 29: **end for**

2.2.2 Sphere

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

Algorithm 3 Sphere Creation

num_hills \leftarrow 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

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

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

Chapter 4

Signal Simulation

Overview

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

4.1 Transmitted Signal

- In probing systems, which are systems which transmit a signal and infer qualitative and quantitative characteristics 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 to the Dirac-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 signal 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 \leftarrow
ScatterReflectivity \leftarrow
AngleDensity \leftarrow Quantization of angles per degree.
AzimuthalBeamwidth \leftarrow Azimuthal Beamwidth
RangeCellWidth \leftarrow 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 others in the following range cells.

Algorithm 5 Range Histogram Method

ScatterCoordinates \leftarrow
ScatterReflectivity \leftarrow
AngleDensity \leftarrow Quantization of angles per degree.
AzimuthalBeamwidth \leftarrow Azimuthal Beamwidth
RangeCellWidth \leftarrow 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 \leftarrow
ScatterReflectivity \leftarrow
AngleDensity \leftarrow Quantization of angles per degree.
AzimuthalBeamwidth \leftarrow Azimuthal Beamwidth
RangeCellWidth \leftarrow 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 infinitesimal 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 direct 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 remains 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 signal'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 \leftarrow
ScatterReflectivity \leftarrow
AngleDensity \leftarrow Quantization of angles per degree.
AzimuthalBeamwidth \leftarrow Azimuthal Beamwidth
RangeCellWidth \leftarrow 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 \leftarrow
ScatterReflectivity \leftarrow
AngleDensity \leftarrow Quantization of angles per degree.
AzimuthalBeamwidth \leftarrow Azimuthal Beamwidth
RangeCellWidth \leftarrow 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 \leftarrow
ScatterReflectivity \leftarrow
AngleDensity \leftarrow Quantization of angles per degree.
AzimuthalBeamwidth \leftarrow Azimuthal Beamwidth
RangeCellWidth \leftarrow 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 artificial 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, *createArtificialAcousticImage*. This function calls the class ULA's method appropriately.

3. After the ULA's create their respective acoustic images, they are put together, either by dimension-wise concatenation or depth-wise concatenation and feed to the neural net to produce control sequences.
4. We need to work on the dimensions of these images though. The best thing to do right now is to finalize the transmitter and receiver parameters and then over-estimate the dimensions of the final beamforming-produced image. We shall then use these dimensions to create the artificial acoustic image and start training the policy.

Algorithm 10 Artifical Acoustic Imaging

ScatterCoordinates \leftarrow Coordinates of points in the point-cloud.

auvCoordinates \leftarrow Coordinates of AUV/ULA.

Chapter 7

Results

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.

```

1 // header-files
2 #include <iostream>
3 #include <ostream>
4 #include <torch/torch.h>
5
6 #pragma once
7
8 // hash defines
9 #ifndef PRINTSPACE
10 #define PRINTSPACE      std::cout<<"\n\n\n\n\n\n\n\n"<<std::endl;
11 #endif
12 #ifndef PRINTSMALLLINE
13 #define PRINTSMALLLINE std::cout<<"-----"<<std::endl;
14 #endif
15 #ifndef PRINTLINE
16 #define PRINTLINE      std::cout<<"======"<<std::endl;
17 #endif
18 #ifndef DEVICE
19     #define DEVICE      torch::kMPS
20     // #define DEVICE    torch::kCPU
21 #endif
22
23
24 #define PI              3.14159265
25
26
27 // function to print tensor size
28 void print_tensor_size(const torch::Tensor& inputTensor) {
29     // Printing size
30     std::cout << "[";
31     for (const auto& size : inputTensor.sizes()) {
32         std::cout << size << ",";
33     }

```

```

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.

```

1  // header-files
2  #include <iostream>
3  #include <ostream>
4  #include <cmath>
5
6  // Including classes
7  #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/ScattererClass.h"
8
9  // Including functions
10 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fCart2Sph.cpp"
11 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fPrintTensorSize.cpp"
12 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fSph2Cart.cpp"
13
14 #pragma once
15
16 // hash defines
17 #ifndef PRINTSPACE
18 #   define PRINTSPACE    std::cout<<"\n\n\n\n\n\n\n\n\n\n"<<std::endl;
19 #endif
20 #ifndef PRINTSMALLLINE
21 #   define PRINTSMALLLINE std::cout<<"-----"<<std::endl;
22 #endif
23 #ifndef PRINTLINE
24 #   define PRINTLINE      std::cout<<"===== "<<std::endl;
25 #endif
26
27 #define PI                3.14159265
28 #define DEBUGMODE_TRANSMITTER    false
29
30 #ifndef DEVICE
31     #define DEVICE        torch::kMPS
32     // #define DEVICE      torch::kCPU
33 #endif
34
35
36
37 // control panel
38 #define ENABLE_RAYTRACING    false
39
40
41
42
43
44
45
46
47 class TransmitterClass{
48 public:
49
50     // physical/intrinsic properties
51     torch::Tensor location;          // location tensor
52     torch::Tensor pointing_direction; // pointing direction
53
54     // basic parameters
55     torch::Tensor Signal;           // transmitted signal (LFM)
56     float azimuthal_angle;          // transmitter's azimuthal pointing direction
57     float elevation_angle;          // transmitter's elevation pointing direction
58     float azimuthal_beamwidth;      // azimuthal beamwidth of transmitter
59     float elevation_beamwidth;      // elevation beamwidth of transmitter
60     float range;                    // a parameter used for spotlight mode.
61
62     // transmitted signal attributes
63     float f_low;                    // lowest frequency of LFM
64     float f_high;                   // highest frequency of LFM
65     float fc;                       // center frequency of LFM
66     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,
162     // std::cout<<"\t TransmitterClass: AUV_pointing_vector_spherical =
163     // std::cout<<"\t TransmitterClass: AUV_pointing_vector_spherical =
164     // calculating azimuth and elevation of transmitter object
165     torch::Tensor absolute_azimuth_of_transmitter = yaw +
166     torch::tensor({this->azimuthal_angle}).to(torch::kFloat).to(DEVICE);
167     torch::Tensor absolute_elevation_of_transmitter = pitch +
168     torch::tensor({this->elevation_angle}).to(torch::kFloat).to(DEVICE);
169     if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 149 \n";
170
171     // std::cout<<"\t TransmitterClass: this->azimuthal_angle = "<<this->azimuthal_angle<<std::endl;
172     // std::cout<<"\t TransmitterClass: this->elevation_angle = "<<this->elevation_angle<<std::endl;
173     // std::cout<<"\t TransmitterClass: absolute_azimuth_of_transmitter =
174     // std::cout<<"\t TransmitterClass: absolute_elevation_of_transmitter =
175     // converting back to Cartesian
176     torch::Tensor pointing_direction_spherical = torch::zeros({3,1}).to(torch::kFloat).to(DEVICE);
177     pointing_direction_spherical[0] = absolute_azimuth_of_transmitter;
178     pointing_direction_spherical[1] = absolute_elevation_of_transmitter;
179     pointing_direction_spherical[2] = torch::tensor({1}).to(torch::kFloat).to(DEVICE);
180     if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 60 \n";
181
182     this->pointing_direction = fSph2Cart(pointing_direction_spherical);
183     if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 169 \n";
184
185 }
186
187 /*=====
188 Aim: Subsetting Scatterers inside the cone
189 -----*/
190 steps:
191 1. Find azimuth and range of all points.
192 2. Find azimuth and range of current pointing vector.
193 3. Subtract azimuth and range of points from that of azimuth and range of current pointing vector
194 4. Use tilted ellipse equation to find points in the ellipse
195 -----*/
196 void subsetScatterers(ScattererClass* scatterers,
197 float tilt_angle){
198
199     // translationally change origin
200     scatterers->coordinates = \
201     scatterers->coordinates - this->location;
202
203     /*
204     Note: I think something we can do is see if we can subset the matrices by checking coordinate values
205     right away. If one of the coordinate values is x (relative coordinates), we know for sure that
206     the distance is greater than x, for sure. So, maybe that's something that we can work with

```

```

204 */
205
206 // Finding spherical coordinates of scatterers and pointing direction
207 torch::Tensor scatterers_spherical = fCart2Sph(scatterers->coordinates);
208 torch::Tensor pointing_direction_spherical = fCart2Sph(this->pointing_direction);
209
210 // Calculating relative azimuths and radians
211 torch::Tensor relative_spherical = \
212     scatterers_spherical - pointing_direction_spherical;
213
214 // clearing some stuff up
215 scatterers_spherical.reset();
216 pointing_direction_spherical.reset();
217
218 // tensor corresponding to switch.
219 torch::Tensor tilt_angle_Tensor = \
220     torch::tensor({tilt_angle}).to(torch::kFloat).to(DEVICE);
221
222 // calculating length of axes
223 torch::Tensor axis_a = \
224     torch::tensor({
225         this->azimuthal_beamwidth / 2
226     }).to(torch::kFloat).to(DEVICE);
227 torch::Tensor axis_b = \
228     torch::tensor({
229         this->elevation_beamwidth / 2
230     }).to(torch::kFloat).to(DEVICE);
231
232 // part of calculating the tilted ellipse
233 torch::Tensor xcosa = relative_spherical[0] * torch::cos(tilt_angle_Tensor * PI/180);
234 torch::Tensor ysina = relative_spherical[1] * torch::sin(tilt_angle_Tensor * PI/180);
235 torch::Tensor xsina = relative_spherical[0] * torch::sin(tilt_angle_Tensor * PI/180);
236 torch::Tensor ycosa = relative_spherical[1] * torch::cos(tilt_angle_Tensor * PI/180);
237 relative_spherical.reset();
238
239 // finding points inside the tilted ellipse
240 torch::Tensor scatter_boolean = \
241     torch::div(torch::square(xcosa + ysina), torch::square(axis_a)) + \
242     torch::div(torch::square(xsina - ycosa), torch::square(axis_b)) <= 1;
243
244 // clearing
245 xcosa.reset(); ysina.reset(); xsina.reset(); ycosa.reset();
246
247 // subsetting points within the elliptical beam
248 auto mask = (scatter_boolean == 1); // creating a mask
249 scatterers->coordinates = scatterers->coordinates.index({torch::indexing::Slice(), mask});
250 scatterers->reflectivity = scatterers->reflectivity.index({torch::indexing::Slice(), mask});
251
252 // this is where histogram shadowing comes in (later)
253 if (ENABLE_RAYTRACING) {rangeHistogramShadowing(scatterers); std::cout<<"\t\t TransmitterClass: line
    232 "<<std::endl;}
254
255 // translating back to the points
256 scatterers->coordinates = scatterers->coordinates + this->location;
257
258 }
259
260 /*****
261 Aim: Shadowing method (range-histogram shadowing)
262 .....
263 Note:
264 > cut down the number of threads into range-cells
265 > for each range cell, calculate histogram
266 >
267 std::cout<<"\t TransmitterClass: "
268 -----*/
269 void rangeHistogramShadowing(ScattererClass* scatterers){
270
271     // converting points to spherical coordinates
272     torch::Tensor spherical_coordinates = fCart2Sph(scatterers->coordinates); std::cout<<"\t\t
    TransmitterClass: line 252 "<<std::endl;
273
274     // finding maximum range

```

```

275 torch::Tensor maxdistanceofpoints = torch::max(spherical_coordinates[2]); std::cout<<"\t\t
    TransmitterClass: line 256 "<<std::endl;
276
277 // calculating number of range-cells (verified)
278 int numrangecells = std::ceil(maxdistanceofpoints.item<int>()/this->rangeQuantSize);
279
280 // finding range-cell boundaries (verified)
281 torch::Tensor rangeBoundaries = \
282     torch::linspace(this->rangeQuantSize, \
283         numrangecells * this->rangeQuantSize, \
284         numrangecells); std::cout<<"\t\t TransmitterClass: line 263 "<<std::endl;
285
286 // creating the checkbox (verified)
287 int numazimuthcells = std::ceil(this->azimuthal_beamwidth * this->azimuthQuantDensity);
288 int numelevationcells = std::ceil(this->elevation_beamwidth * this->elevationQuantDensity);
289     std::cout<<"\t\t TransmitterClass: line 267 "<<std::endl;
290
291 // finding the deltas
292 float delta_azimuth = this->azimuthal_beamwidth / numazimuthcells;
293 float delta_elevation = this->elevation_beamwidth / numelevationcells; std::cout<<"\t\t
    TransmitterClass: line 271"<<std::endl;
294
295 // creating the centers (verified)
296 torch::Tensor azimuth_centers = torch::linspace(delta_azimuth/2, \
297     numazimuthcells * delta_azimuth - delta_azimuth/2, \
298     numazimuthcells);
299 torch::Tensor elevation_centers = torch::linspace(delta_elevation/2, \
300     numelevationcells * delta_elevation - delta_elevation/2, \
301     numelevationcells); std::cout<<"\t\t TransmitterClass:
    line 279"<<std::endl;
302
303 // centering (verified)
304 azimuth_centers = azimuth_centers + torch::tensor({this->azimuthal_angle - \
305     (this->azimuthal_beamwidth/2)}).to(torch::kFloat);
306 elevation_centers = elevation_centers + torch::tensor({this->elevation_angle - \
307     (this->elevation_beamwidth/2)}).to(torch::kFloat);
308     std::cout<<"\t\t TransmitterClass: line
309     285"<<std::endl;
310
311 // building checkboxes
312 torch::Tensor checkbox = torch::zeros({numelevationcells, numazimuthcells}, torch::kBool);
313 torch::Tensor finalScatterBox = torch::zeros({numelevationcells, numazimuthcells,
314     3}).to(torch::kFloat);
315 torch::Tensor finalReflectivityBox = torch::zeros({numelevationcells,
316     numazimuthcells}).to(torch::kFloat); std::cout<<"\t\t TransmitterClass: line 290"<<std::endl;
317
318 // going through each-range-cell
319 for(int i = 0; i<(int)rangeBoundaries.numel(); ++i){
320     this->internal_subsetCurrentRangeCell(rangeBoundaries[i], \
321         scatterers, \
322         checkbox, \
323         finalScatterBox, \
324         finalReflectivityBox, \
325         azimuth_centers, \
326         elevation_centers, \
327         spherical_coordinates); std::cout<<"\t\t TransmitterClass: line
328     301"<<std::endl;
329
330 // after each-range-cell
331 torch::Tensor checkboxfilled = torch::sum(checkbox);
332 std::cout<<"\t\t\t\t checkbox-filled = "<<checkboxfilled.item<int>()/checkbox.numel()<<" |
333     percent = "<<100 * checkboxfilled.item<float>()/(float)checkbox.numel()<<std::endl;
334
335 }
336
337 // converting from box structure to [3, num-points] structure
338 torch::Tensor final_coords_spherical = \
339     torch::permute(finalScatterBox, {2, 0, 1}).reshape({3, (int)(finalScatterBox.numel()/3)});
340 torch::Tensor final_coords_cart = fSph2Cart(final_coords_spherical); std::cout<<"\t\t
    TransmitterClass: line 308"<<std::endl;
341 std::cout<<"\t\t finalReflectivityBox.shape = "; fPrintTensorSize(finalReflectivityBox);
342 torch::Tensor final_reflectivity = finalReflectivityBox.reshape({finalReflectivityBox.numel()});
343     std::cout<<"\t\t TransmitterClass: line 310"<<std::endl;

```

```

336 torch::Tensor test_checkbox = checkbox.reshape({checkbox.numel()}); std::cout<<"\t\t TransmitterClass:
    line 311"<<std::endl;
337
338 // just taking the points corresponding to the filled. Else, there's gonna be a lot of zero zero zero
    tensors
339 auto mask = (test_checkbox == 1); std::cout<<"\t\t TransmitterClass: line 319"<<std::endl;
340 final_coords_cart = final_coords_cart.index({torch::indexing::Slice(), mask}); std::cout<<"\t\t
    TransmitterClass: line 320"<<std::endl;
341 final_reflectivity = final_reflectivity.index({mask}); std::cout<<"\t\t TransmitterClass: line
    321"<<std::endl;
342
343 // overwriting the scatterers
344 scatterers->coordinates = final_coords_cart;
345 scatterers->reflectivity = final_reflectivity; std::cout<<"\t\t TransmitterClass: line 324"<<std::endl;
346
347 }
348
349
350 void internal_subsetCurrentRangeCell(torch::Tensor rangeupperlimit, \
351                                     ScattererClass* scatterers, \
352                                     torch::Tensor& checkbox, \
353                                     torch::Tensor& finalScatterBox, \
354                                     torch::Tensor& finalReflectivityBox, \
355                                     torch::Tensor& azimuth_centers, \
356                                     torch::Tensor& elevation_centers, \
357                                     torch::Tensor& spherical_coordinates){
358
359 // finding indices for points in the current range-cell
360 torch::Tensor pointsincurrentrangeCell = \
361     torch::mul((spherical_coordinates[2] <= rangeupperlimit) , \
362               (spherical_coordinates[2] > rangeupperlimit - this->rangeQuantSize));
363
364 // checking out if there are no points in this range-cell
365 int num311 = torch::sum(pointsincurrentrangeCell).item<int>();
366 if(num311 == 0) return;
367
368 // calculating delta values
369 float delta_azimuth = azimuth_centers[1].item<float>() - azimuth_centers[0].item<float>();
370 float delta_elevation = elevation_centers[1].item<float>() - elevation_centers[0].item<float>();
371
372 // subsetting points in the current range-cell
373 auto mask = (pointsincurrentrangeCell == 1); // creating a mask
374 torch::Tensor reflectivityincurrentrangeCell =
    scatterers->reflectivity.index({torch::indexing::Slice(), mask});
375 pointsincurrentrangeCell = spherical_coordinates.index({torch::indexing::Slice(),
    mask});
376
377 // finding number of azimuth sizes and what not
378 int numazimuthcells = azimuth_centers.numel();
379 int numelevationcells = elevation_centers.numel();
380
381 // go through all the combinations
382 for(int azi_index = 0; azi_index < numazimuthcells; ++azi_index){
383     for(int ele_index = 0; ele_index < numelevationcells; ++ele_index){
384
385         // check if this particular azimuth-elevation direction has been taken-care of.
386         if (checkbox[ele_index][azi_index].item<bool>()) break;
387
388         // init (verified)
389         torch::Tensor current_azimuth = azimuth_centers.index({azi_index});
390         torch::Tensor current_elevation = elevation_centers.index({ele_index});
391
392         // // finding azimuth boolean
393         // torch::Tensor azi_neighbours = torch::abs(pointsincurrentrangeCell[0] - current_azimuth);
394         // azi_neighbours = azi_neighbours <= delta_azimuth; // tinker with this.
395
396         // // finding elevation boolean
397         // torch::Tensor ele_neighbours = torch::abs(pointsincurrentrangeCell[1] - current_elevation);
398         // ele_neighbours = ele_neighbours <= delta_elevation;
399
400         // finding azimuth boolean
401         torch::Tensor azi_neighbours = torch::abs(pointsincurrentrangeCell[0] - current_azimuth);
402         azi_neighbours = azi_neighbours <= this->azimuthShadowThreshold; // tinker with

```

```

    this.

```

```

403
404 // finding elevation boolean
405 torch::Tensor ele_neighbours = torch::abs(pointsincurrentrange[1] - current_elevation);
406 ele_neighbours
407         = ele_neighbours <= this->elevationShadowThreshold;
408
409 // combining booleans: means find all points that are within the limits of both the azimuth and
    boolean.
410 torch::Tensor neighbours_boolean = torch::mul(azi_neighbours, ele_neighbours);
411
412 // checking if there are any points along this direction
413 int num347 = torch::sum(neighbours_boolean).item<int>();
414 if (num347 == 0) continue;
415
416 // findings point along this direction
417 mask = (neighbours_boolean == 1);
418 torch::Tensor coords_along_aziele_spherical =
    pointsincurrentrange.index({torch::indexing::Slice(), mask});
419 torch::Tensor reflectivity_along_aziele =
    reflectivityincurrentrange.index({torch::indexing::Slice(), mask});
420
421 // finding the index where the points are at the maximum distance
422 int index_where_min_range_is = torch::argmin(coords_along_aziele_spherical[2]).item<int>();
423 torch::Tensor closest_coord = coords_along_aziele_spherical.index({torch::indexing::Slice(), \
424         index_where_min_range_is});
425 torch::Tensor closest_reflectivity = reflectivity_along_aziele.index({torch::indexing::Slice(),
    \
426         index_where_min_range_is});
427
428 // filling the matrices up
429 finalScatterBox.index_put_({ele_index, azi_index, torch::indexing::Slice()}, \
430         closest_coord.reshape({1,1,3}));
431 finalReflectivityBox.index_put_({ele_index, azi_index}, \
432         closest_reflectivity);
433 checkbox.index_put_({ele_index, azi_index}, \
434         true);
435
436     }
437 }
438 }
439
440
441
442
443 };

```

8.1.3 Class: Uniform Linear Array

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

```

1  // bringing in the header files
2  #include <cstdint>
3  #include <iostream>
4  #include <ostream>
5  #include <stdexcept>
6  #include <torch/torch.h>
7  #include <omp.h>           // the openMP
8
9
10 // class definitions
11 #include "ScattererClass.h"
12 #include "TransmitterClass.h"
13
14 // bringing in the functions
15 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fPrintTensorSize.cpp"
16 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fConvolveColumns.cpp"
17 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fBuffer2D.cpp"
18 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fConvolve1D.cpp"
19 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fCart2Sph.cpp"
20
21 #pragma once
22
23 // hash defines
24 #ifndef PRINTSPACE
25     #define PRINTSPACE    std::cout<<"\n\n\n\n\n\n\n\n"<<std::endl;
26 #endif
27 #ifndef PRINTSMALLLINE
28     #define PRINTSMALLLINE
29         std::cout<<"-----"<<std::endl;
30 #endif
31 #ifndef PRINTLINE
32     #define PRINTLINE
33         std::cout<<"===== "<<std::endl;
34 #endif
35 #ifndef PRINTDOTS
36     #define PRINTDOTS
37         std::cout<<"..... "<<std::endl;
38 #endif
39
40 #ifndef DEVICE
41     // #define DEVICE    torch::kMPS
42     #define DEVICE    torch::kCPU
43 #endif
44
45 #define PI    3.14159265
46 #define COMPLEX_1j    torch::complex(torch::zeros({1}), torch::ones({1}))
47
48 // #define DEBUG_ULA true
49 #define DEBUG_ULA false
50
51 class ULAClass{
52 public:
53     // intrinsic parameters
54     int num_sensors;           // number of sensors
55     float inter_element_spacing; // space between sensors
56     torch::Tensor coordinates; // coordinates of each sensor
57     float sampling_frequency; // sampling frequency of the sensors
58     float recording_period; // recording period of the ULA
59     torch::Tensor location; // location of first coordinate
60
61     // derived stuff
62     torch::Tensor sensorDirection;
63     torch::Tensor signalMatrix;

```



```

64
65 // decimation-related
66 int decimation_factor; // the new decimation factor
67 float post_decimation_sampling_frequency; // the new sampling frequency
68 torch::Tensor lowpassFilterCoefficientsForDecimation; //
69
70 // imaging related
71 float range_resolution; // theoretical range-resolution =  $\frac{c}{2B}$ 
72 float azimuthal_resolution; // theoretical azimuth-resolution =
     $\frac{\lambda}{(N-1) \cdot \text{inter-element-distance}}$ 
73 float range_cell_size; // the range-cell quanta we're choosing for efficiency trade-off
74 float azimuth_cell_size; // the azimuth quanta we're choosing
75 torch::Tensor mulFFTMATRIX; // the matrix containing the delays for each-element as a slot
76 torch::Tensor azimuth_centers; // tensor containing the azimuth centers
77 torch::Tensor range_centers; // tensor containing the range-centers
78 int frame_size; // the frame-size corresponding to a range cell in a decimated signal
    matrix
79 torch::Tensor matchFilter; // torch tensor containing the match-filter
80 int num_buffer_zeros_per_frame; // number of zeros we're adding per frame to ensure no-rotation
81 torch::Tensor beamformedImage; // the beamformed image
82 torch::Tensor cartesianImage;
83
84 // artificial acoustic-image related
85 torch::Tensor currentArtificialAcousticImage; // the acoustic image directly produced
86
87 // constructor
88 ULAClass(int numsensors = 32,
89 float inter_element_spacing = 1e-3,
90 torch::Tensor coordinates = torch::zeros({3, 2}),
91 float sampling_frequency = 48e3,
92 float recording_period = 1,
93 torch::Tensor location = torch::zeros({3,1}),
94 torch::Tensor signalMatrix = torch::zeros({1, 32}),
95 torch::Tensor lowpassFilterCoefficientsForDecimation = torch::zeros({1,10})):
96 num_sensors(numsensors),
97 inter_element_spacing(inter_element_spacing),
98 coordinates(coordinates),
99 sampling_frequency(sampling_frequency),
100 recording_period(recording_period),
101 location(location),
102 signalMatrix(signalMatrix),
103 lowpassFilterCoefficientsForDecimation(lowpassFilterCoefficientsForDecimation){
104 // calculating ULA direction
105 torch::Tensor sensorDirection = coordinates.slice(1, 0, 1) - coordinates.slice(1, 1, 2);
106
107 // normalizing
108 float normvalue = torch::linalg_norm(sensorDirection, 2, 0, true, torch::kFloat).item<float>();
109 if (normvalue != 0){
110     sensorDirection = sensorDirection / normvalue;
111 }
112
113 // copying direction
114 this->sensorDirection = sensorDirection;
115 }
116
117 // overriding printing
118 friend std::ostream& operator<<(std::ostream& os, ULAClass& ula){
119     os<<"\t number of sensors : "<<ula.num_sensors <<std::endl;
120     os<<"\t inter-element spacing: "<<ula.inter_element_spacing <<std::endl;
121     os<<"\t sensor-direction " <<torch::transpose(ula.sensorDirection, 0, 1)<<std::endl;
122     PRINTSMALLLINE
123     return os;
124 }
125
126 /* =====
127 Aim: Init
128 ===== */
129 void init(TransmitterClass* transmitterObj){
130
131     // calculating range-related parameters
132     this->range_resolution = 1500/(2 * transmitterObj->fc);
133     this->range_cell_size = 40 * this->range_resolution;
134

```



```

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

```

```

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

```

```

279         (int)(scatterers->coordinates.numel()/3),          \
280         3});
281
282     // repeating along the y-direction number of sensor times.
283     slottedCoordinates =
284         torch::tile(slottedCoordinates,                    \
285                     {this->num_sensors, 1, 1});
286     std::vector<int64_t> slottedCoordinates_shape =          \
287         slottedCoordinates.sizes().vec();
288
289     // finding the shape of the sensor-coordinates
290     std::vector<int64_t> sensor_coordinates_shape = \
291         this->coordinates.sizes().vec();
292
293     // creating a slot tensor out of the sensor-coordinates
294     torch::Tensor slottedSensors =                        \
295         torch::permute(this->coordinates.reshape({
296             sensor_coordinates_shape[0],                    \
297             sensor_coordinates_shape[1],                    \
298             1}), {2, 1, 0}).reshape({(int)(this->coordinates.numel()/3), \
299                                     1,                      \
300                                     3});
301
302     // repeating slices along the x-coordinates
303     slottedSensors =                                     \
304         torch::tile(slottedSensors,                      \
305                     {1, slottedCoordinates_shape[1], 1});
306
307     // slotting the coordinate of the transmitter and duplicating along dimensions [0] and [1]
308     torch::Tensor slotted_location =                    \
309         torch::permute(this->location.reshape({3, 1, 1}), \
310                       {2, 1, 0}).reshape({1,1,3});
311     slotted_location =                                  \
312         torch::tile(slotted_location, {slottedCoordinates_shape[0], \
313                                         slottedCoordinates_shape[1], \
314                                         1});
315
316
317     // subtracting to find the relative distances
318     torch::Tensor distBetweenScatterersAndSensors =      \
319         torch::linalg_norm(slottedCoordinates - slottedSensors, \
320                             2, 2, true, torch::kFloat);
321
322     // subtracting distance between relative fields
323     torch::Tensor distBetweenScatterersAndTransmitter = \
324         torch::linalg_norm(slottedCoordinates - slotted_location, \
325                             2, 2, true, torch::kFloat);
326
327     // adding up the distances
328     torch::Tensor distOfFlight = \
329         distBetweenScatterersAndSensors + distBetweenScatterersAndTransmitter;
330     torch::Tensor timeOfFlight = distOfFlight/1500;
331     torch::Tensor samplesOfFlight = \
332         torch::floor(timeOfFlight.squeeze() \
333                     * this->sampling_frequency);
334
335
336     // Adding pulses
337     #pragma omp parallel for
338     for(int sensor_index = 0; sensor_index < this->num_sensors; ++sensor_index){
339         for(int scatter_index = 0; scatter_index < samplesOfFlight[0].numel(); ++scatter_index){
340
341             // getting the sample where the current scatter's contribution must be placed.
342             int where_to_place = \
343                 samplesOfFlight.index({sensor_index, \
344                                         scatter_index \
345                                         }).item<int>();
346
347             // checking whether that point is out of bounds
348             if(where_to_place >= numsamples) continue;
349
350             // placing a reflectivity-scaled impulse in there
351             this->signalMatrix.index_put_({where_to_place, sensor_index}, \

```

```

352         this->signalMatrix.index({where_to_place, \
353                                 sensor_index}) + \
354         scatterers->reflectivity.index({0, \
355                                         scatter_index}));
356     }
357 }
358
359
360 // // Adding pulses
361 // for(int sensor_index = 0; sensor_index < this->num_sensors; ++sensor_index){
362
363     // // indices associated with current index
364     // torch::Tensor tensor_containing_placing_indices = \
365     //     samplesOfFlight[sensor_index].to(torch::kInt);
366
367     // // calculating histogram
368     // auto uniqueOutputs = at::_unique(tensor_containing_placing_indices, false, true);
369     // torch::Tensor bruh = std::get<1>(uniqueOutputs);
370     // torch::Tensor uniqueValues = std::get<0>(uniqueOutputs).to(torch::kInt);
371     // torch::Tensor uniqueCounts = torch::bincount(bruh).to(torch::kInt);
372
373     // // placing values according to histogram
374     // this->signalMatrix.index_put_({uniqueValues.to(torch::kLong), sensor_index}, \
375     //     uniqueCounts.to(torch::kFloat));
376
377 // }
378
379 // Creating matrix out of transmitted signal
380 torch::Tensor signalTensorAsArgument = \
381     transmitterObj->Signal.reshape({
382         transmitterObj->Signal.numel(), \
383         1});
384 signalTensorAsArgument = \
385     torch::tile(signalTensorAsArgument, \
386         {1, this->signalMatrix.size(1)});
387
388
389 // convolving the pulse-matrix with the signal matrix
390 fConvolveColumns(this->signalMatrix, \
391     signalTensorAsArgument);
392
393
394 // trimming the convolved signal since the signal matrix length remains the same
395 this->signalMatrix = \
396     this->signalMatrix.index({
397         torch::indexing::Slice(0, numsamples), \
398         torch::indexing::Slice()});
399
400 // returning
401 return;
402 }
403
404 /* =====
405 Aim: Decimating basebanded-received signal
406 ----- */
407 void nfdc_decimateSignal(TransmitterClass* transmitterObj){
408
409     // creating the matrix for frequency-shifting
410     torch::Tensor integerArray = torch::linspace(0, \
411         this->signalMatrix.size(0)-1, \
412         this->signalMatrix.size(0)).reshape({this->signalMatrix.size(0),
413         1});
414     integerArray = torch::tile(integerArray, {1, this->num_sensors});
415     integerArray = torch::exp(COMPLEX_1j * transmitterObj->fc * integerArray);
416
417     // storing original number of samples
418     int original_signalMatrix_numsamples = this->signalMatrix.size(0);
419
420     // producing frequency-shifting
421     this->signalMatrix = torch::mul(this->signalMatrix, integerArray);
422
423     // low-pass filter
424     torch::Tensor lowpassfilter_impulseresponse = \

```

```

424         this->lowpassFilterCoefficientsForDecimation.reshape(      \
425             {this->lowpassFilterCoefficientsForDecimation.numel(), \
426                 1});
427     lowpassfilter_impulseresponse =                                \
428         torch::tile(lowpassfilter_impulseresponse,                  \
429             {1, this->signalMatrix.size(1)});
430
431     // low-pass filtering the signal
432     fConvolveColumns(this->signalMatrix,
433         lowpassfilter_impulseresponse);
434
435     // Cutting down the extra-samples from convolution
436     this->signalMatrix = \
437         this->signalMatrix.index({torch::indexing::Slice(0, original_signalMatrix_numsamples), \
438             torch::indexing::Slice()});
439
440     // // Cutting off samples in the front.
441     // int cutoffpoint = lowpassfilter_impulseresponse.size(0) - 1;
442     // this->signalMatrix = \
443     //     this->signalMatrix.index({ \
444     //         torch::indexing::Slice(cutoffpoint, \
445     //             torch::indexing::None), \
446     //         torch::indexing::Slice() \
447     //     });
448
449     // building parameters for downsampling
450     int decimation_factor = std::floor(this->sampling_frequency/transmitterObj->bandwidth);
451     this->decimation_factor = decimation_factor;
452     this->post_decimation_sampling_frequency = \
453         this->sampling_frequency / this->decimation_factor;
454     int numsamples_after_decimation = std::floor(this->signalMatrix.size(0)/decimation_factor);
455
456     // building the samples which will be subsetted
457     torch::Tensor samplingIndices = \
458         torch::linspace(0, \
459             numsamples_after_decimation * decimation_factor - 1, \
460             numsamples_after_decimation).to(torch::kInt);
461
462     // downsampling the low-pass filtered signal
463     this->signalMatrix = \
464         this->signalMatrix.index({samplingIndices, \
465             torch::indexing::Slice()});
466
467     // returning
468     return;
469 }
470
471 /* =====
472 Aim: Match-filtering
473 ----- */
474 void nfdc_matchFilterDecimatedSignal(){
475
476     // Creating a 2D matrix out of the signal
477     torch::Tensor matchFilter2DMatrix = \
478         this->matchFilter.reshape({this->matchFilter.numel(), 1});
479     matchFilter2DMatrix = \
480         torch::tile(matchFilter2DMatrix, \
481             {1, this->num_sensors});
482
483
484     // 2D convolving to produce the match-filtering
485     fConvolveColumns(this->signalMatrix, \
486         matchFilter2DMatrix);
487
488
489     // Trimming the signal to contain just the signals that make sense to us
490     int startingpoint = matchFilter2DMatrix.size(0) - 1;
491     this->signalMatrix = \
492         this->signalMatrix.index({ \
493             torch::indexing::Slice(startingpoint, \
494                 torch::indexing::None), \
495             torch::indexing::Slice()});
496

```

```

497 // // trimming the two ends of the signal
498 // int startingpoint = matchFilter2DMatrix.size(0) - 1;
499 // int endingpoint = this->signalMatrix.size(0) \
500 // - matchFilter2DMatrix.size(0) \
501 // + 1;
502 // this->signalMatrix = \
503 // this->signalMatrix.index({ \
504 // torch::indexing::Slice(startingpoint, \
505 // endingpoint), \
506 // torch::indexing::Slice()});
507
508
509 }
510
511 /* =====
512 Aim: precompute delay-matrices
513 ===== */
514 void nfdc_precomputeDelayMatrices(TransmitterClass* transmitterObj){
515
516 // calculating range-related parameters
517 int number_of_range_cells = \
518 std::ceil(((this->recording_period * 1500)/2)/this->range_cell_size);
519 int number_of_azimuths_to_image = \
520 std::ceil(transmitterObj->azimuthal_beamwidth / this->azimuth_cell_size);
521
522 // creating centers of range-cell centers
523 torch::Tensor range_centers = \
524 this->range_cell_size * \
525 torch::linspace(0, \
526 number_of_range_cells-1, \
527 number_of_range_cells).to(torch::kFloat) + \
528 this->range_cell_size/2;
529 this->range_centers = range_centers;
530
531 // creating discretized azimuth-centers
532 torch::Tensor azimuth_centers = \
533 this->azimuth_cell_size * \
534 torch::linspace(0, \
535 number_of_azimuths_to_image - 1, \
536 number_of_azimuths_to_image) + \
537 this->azimuth_cell_size/2;
538 this->azimuth_centers = azimuth_centers;
539
540 // finding the mesh values
541 auto range_azimuth_meshgrid = \
542 torch::meshgrid({range_centers, azimuth_centers}, "ij");
543 torch::Tensor range_grid = range_azimuth_meshgrid[0]; // the columns are range_centers
544 torch::Tensor azimuth_grid = range_azimuth_meshgrid[1]; // the rows are azimuth-centers
545
546 // going from 2D to 3D
547 range_grid = torch::tile(range_grid.reshape({range_grid.size(0), \
548 range_grid.size(1), \
549 1}), \
550 {1,1,this->num_sensors});
551 azimuth_grid = torch::tile(azimuth_grid.reshape({azimuth_grid.size(0), \
552 azimuth_grid.size(1), \
553 1}), \
554 {1, 1, this->num_sensors});
555
556 // creating x_m tensor
557 torch::Tensor sensorCoordinatesSlot = \
558 this->inter_element_spacing * \
559 torch::linspace(0, \
560 this->num_sensors - 1, \
561 this->num_sensors).reshape({1,1,this->num_sensors}).to(torch::kFloat);
562 sensorCoordinatesSlot = \
563 torch::tile(sensorCoordinatesSlot, \
564 {range_grid.size(0), \
565 range_grid.size(1), \
566 1});
567 if(DEBUG_ULA)
568 std::cout << "\t sensorCoordinatesSlot.sizes().vec() = " \
569 << sensorCoordinatesSlot.sizes().vec() \

```

```

570         << std::endl;
571
572     // calculating distances
573     torch::Tensor distanceMatrix = \
574         torch::square(range_grid - sensorCoordinatesSlot) + \
575         torch::mul((2 * sensorCoordinatesSlot), \
576             torch::mul(range_grid, \
577                 1 - torch::cos(azimuth_grid * PI/180)));
578     distanceMatrix = torch::sqrt(distanceMatrix);
579
580     // finding the time taken
581     torch::Tensor timeMatrix = distanceMatrix/1500;
582     torch::Tensor sampleMatrix = timeMatrix * this->sampling_frequency;
583
584     // finding the delay to be given
585     auto bruh390 = torch::max(sampleMatrix, 2, true);
586     torch::Tensor max_delay = std::get<0>(bruh390);
587     torch::Tensor delayMatrix = max_delay - sampleMatrix;
588
589     // now that we have the delay entries, we need to create the matrix that does it
590     int decimation_factor = \
591         std::floor(static_cast<float>(this->sampling_frequency)/transmitterObj->bandwidth);
592     this->decimation_factor = decimation_factor;
593
594
595     // calculating frame-size
596     int frame_size = \
597         std::ceil(static_cast<float>((2 * this->range_cell_size / 1500) * \
598             static_cast<float>(this->sampling_frequency)/decimation_factor));
599     this->frame_size = frame_size;
600
601     // // calculating the buffer-zeros to add
602     // int num_buffer_zeros_per_frame = \
603     //     static_cast<float>(this->num_sensors - 1) * \
604     //     static_cast<float>(this->inter_element_spacing) * \
605     //     this->sampling_frequency / 1500;
606
607     int num_buffer_zeros_per_frame = \
608         std::ceil((this->num_sensors - 1) * \
609             this->inter_element_spacing * \
610             this->sampling_frequency \
611             / (1500 * this->decimation_factor));
612
613     // storing to class member
614     this->num_buffer_zeros_per_frame = \
615         num_buffer_zeros_per_frame;
616
617     // calculating the total frame-size
618     int total_frame_size = \
619         this->frame_size + this->num_buffer_zeros_per_frame;
620
621     // creating the multiplication matrix
622     torch::Tensor mulFFTMMatrix = \
623         torch::linspace(0, \
624             total_frame_size-1, \
625             total_frame_size).reshape({1, \
626                 total_frame_size, \
627                 1}).to(torch::kFloat); // creating an array
628                                     // 1,...,frame_size of shape [1,frame_size, 1];
629
630     mulFFTMMatrix = \
631         torch::div(mulFFTMMatrix, \
632             torch::tensor(total_frame_size).to(torch::kFloat)); // dividing by N
633     mulFFTMMatrix = mulFFTMMatrix * 2 * PI * -1 * COMPLEX_1j; // creating tenosr values for -1j * 2pi * k/N
634     mulFFTMMatrix = \
635         torch::tile(mulFFTMMatrix, \
636             {number_of_range_cells * number_of_azimuths_to_image, \
637                 1, \
638                 this->num_sensors}); // creating the larger tensor for it
639
640     // populating the matrix
641     for(int azimuth_index = 0; \

```

```

641 azimuth_index<number_of_azimuths_to_image; \
642 ++azimuth_index){
643 for(int range_index = 0; \
644     range_index < number_of_range_cells; \
645     ++range_index){
646     // finding the delays for sensors
647     torch::Tensor currentSensorDelays = \
648         delayMatrix.index({range_index, \
649                             azimuth_index, \
650                             torch::indexing::Slice()});
651     // reshaping it to the target size
652     currentSensorDelays = \
653         currentSensorDelays.reshape({1, \
654                                     1, \
655                                     this->num_sensors});
656     // tiling across the plane
657     currentSensorDelays = \
658         torch::tile(currentSensorDelays, \
659                     {1, total_frame_size, 1});
660     // multiplying across the appropriate plane
661     int index_to_place_at = \
662         azimuth_index * number_of_range_cells + \
663         range_index;
664     mulFFTMMatrix.index_put_({index_to_place_at, \
665                             torch::indexing::Slice(), \
666                             torch::indexing::Slice()}, \
667                             currentSensorDelays);
668 }
669 }
670
671 // storing the mulFFTMMatrix
672 this->mulFFTMMatrix = mulFFTMMatrix;
673 }
674
675 /* =====
676 Aim: Beamforming the signal
677 ----- */
678 void nfdc_beamforming(TransmitterClass* transmitterObj){
679
680     // ensuring the signal matrix is in the shape we want
681     if(this->signalMatrix.size(1) != this->num_sensors)
682         throw std::runtime_error("The second dimension doesn't correspond to the number of sensors \n");
683
684     // adding the batch-dimension
685     this->signalMatrix = \
686         this->signalMatrix.reshape({ \
687             1, \
688             this->signalMatrix.size(0), \
689             this->signalMatrix.size(1)});
690
691     // zero-padding to ensure correctness
692     int ideal_length = \
693         std::ceil(this->range_centers.numel() * this->frame_size);
694     int num_zeros_to_pad_signal_along_dimension_0 = \
695         ideal_length - this->signalMatrix.size(1);
696
697     // printing
698     if (DEBUG_ULA) PRINTSMALLLINE
699     if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->range_centers.numel()      =
700         "<<this->range_centers.numel() <<std::endl;
701     if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->frame_size              =
702         "<<this->frame_size <<std::endl;
703     if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | ideal_length                =
704         "<<ideal_length <<std::endl;
705     if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->signalMatrix.size(1)      =
706         "<<this->signalMatrix.size(1) <<std::endl;
707     if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | num_zeros_to_pad_signal_along_dimension_0
708         = "<<num_zeros_to_pad_signal_along_dimension_0 <<std::endl;
709     if (DEBUG_ULA) PRINTSPACE
710
711     // appending or slicing based on the requirements
712     if (num_zeros_to_pad_signal_along_dimension_0 <= 0) {

```



```

708
709 // sending out a warning that slicing is going on
710 if (DEBUG_ULA) std::cerr << "\t\t ULAClass::nfdc_beamforming | Please note that the signal matrix
    has been sliced. This could lead to loss of information" << std::endl;

711
712 // slicing the signal matrix
713 if (DEBUG_ULA) PRINTSPACE
714 if (DEBUG_ULA) std::cout << "\t\t ULAClass::nfdc_beamforming | this->signalMatrix.shape (before
    slicing) = "<< this->signalMatrix.sizes().vec() << std::endl;
715 this->signalMatrix = \
716     this->signalMatrix.index({torch::indexing::Slice(), \
717                             torch::indexing::Slice(0, ideal_length), \
718                             torch::indexing::Slice()});
719 if (DEBUG_ULA) std::cout << "\t\t ULAClass::nfdc_beamforming | this->signalMatrix.shape (after
    slicing) = "<< this->signalMatrix.sizes().vec() << std::endl;
720 if (DEBUG_ULA) PRINTSPACE
721
722 }
723 else {
724     // creating a zero-filled tensor to append to signal matrix
725     torch::Tensor zero_tensor = \
726         torch::zeros({this->signalMatrix.size(0), \
727                     num_zeros_to_pad_signal_along_dimension_0, \
728                     this->num_sensors}).to(torch::kFloat);
729
730     // appending to signal matrix
731     this->signalMatrix = \
732         torch::cat({this->signalMatrix, zero_tensor}, 1);
733 }
734
735 // breaking the signal into frames
736 fBuffer2D(this->signalMatrix, frame_size);
737
738 // add some zeros to the end of frames to accomodate delaying of signals.
739 torch::Tensor zero_filled_tensor = \
740     torch::zeros({this->signalMatrix.size(0), \
741                 this->num_buffer_zeros_per_frame, \
742                 this->num_sensors}).to(torch::kFloat);
743 this->signalMatrix = \
744     torch::cat({this->signalMatrix, \
745                 zero_filled_tensor}, 1);
746
747 // tiling it to ensure that it works for all range-angle combinations
748 int number_of_azimuths_to_image = this->azimuth_centers.numel();
749 this->signalMatrix = \
750     torch::tile(this->signalMatrix, \
751                 {number_of_azimuths_to_image, 1, 1});
752
753 // element-wise multiplying the signals to delay each of the frame accordingly
754 this->signalMatrix = torch::mul(this->signalMatrix, \
755                                this->mulFFTMMatrix);
756
757 // summing up the signals
758 // this->signalMatrix = torch::sum(this->signalMatrix, \
759 //                                2, \
760 //                                true);
761 this->signalMatrix = torch::sum(this->signalMatrix, \
762                                2, \
763                                false);
764
765 // printing some stuff
766 if (DEBUG_ULA) std::cout << "\t\t ULAClass::nfdc_beamforming: this->azimuth_centers.numel() =
    "<< this->azimuth_centers.numel() << std::endl;
767 if (DEBUG_ULA) std::cout << "\t\t ULAClass::nfdc_beamforming: this->range_centers.numel() =
    "<< this->range_centers.numel() << std::endl;
768 if (DEBUG_ULA) std::cout << "\t\t ULAClass::nfdc_beamforming: total number
    "<< this->range_centers.numel() * this->azimuth_centers.numel() << std::endl;
769 if (DEBUG_ULA) std::cout << "\t\t ULAClass::nfdc_beamforming: this->signalMatrix.sizes().vec() =
    "<< this->signalMatrix.sizes().vec() << std::endl;
770
771 // creating a tensor to store the final image
772 torch::Tensor finalImage = \
773     torch::zeros({this->frame_size * this->range_centers.numel(), \

```

```

774         this->azimuth_centers.numel()}).to(torch::kComplexFloat);
775
776     // creating a loop to assign values
777     for(int range_index = 0; range_index < this->range_centers.numel(); ++range_index){
778         for(int angle_index = 0; angle_index < this->azimuth_centers.numel(); ++angle_index){
779
780             // getting row index
781             int rowindex = \
782                 angle_index * this->range_centers.numel() \
783                 + range_index;
784
785             // getting the strip to store
786             torch::Tensor strip = \
787                 this->signalMatrix.index({rowindex, \
788                                         torch::indexing::Slice()});
789
790             // taking just the first few values
791             strip = strip.index({torch::indexing::Slice(0, this->frame_size)});
792
793             // placing the strips on the image
794             finalImage.index_put_({\
795                 torch::indexing::Slice((range_index)*this->frame_size, \
796                                         (range_index+1)*this->frame_size), \
797                 angle_index}, \
798                 strip);
799
800         }
801     }
802
803     // saving the image
804     this->beamformedImage = finalImage;
805
806
807     // converting image from polar to cartesian
808     nfdc_PolarToCartesian();
809     std::cout<<"\t\t ULAClass::nfdc_beamforming: finished nfdc_PolarToCartesian"<<std::endl;
810
811 }
812
813 /* =====
814 Aim: Converting Polar Image to Cartesian
815 .....
816 Note:
817     > For now, we're assuming that the r value is one.
818 ----- */
819 void nfdc_PolarToCartesian(){
820
821
822     // deciding image dimensions
823     int num_pixels_width = 128;
824     int num_pixels_height = 128;
825
826
827     // creating query points
828     torch::Tensor max_right = \
829         torch::cos( \
830             torch::max( \
831                 this->azimuth_centers \
832                 - torch::mean(this->azimuth_centers) \
833                 + torch::tensor({90}).to(torch::kFloat)) \
834             * PI/180);
835     torch::Tensor max_left = \
836         torch::cos( \
837             torch::min(this->azimuth_centers \
838                 - torch::mean(this->azimuth_centers) \
839                 + torch::tensor({90}).to(torch::kFloat)) \
840             * PI/180);
841     torch::Tensor max_top = torch::tensor({1});
842     torch::Tensor max_bottom = torch::min(this->range_centers);
843
844
845     // creating query points along the x-dimension
846     torch::Tensor query_x = \

```

```

847         torch::linspace(                \
848             max_left,                    \
849             max_right,                    \
850             num_pixels_width              \
851         ).to(torch::kFloat);
852
853     torch::Tensor query_y =                \
854         torch::linspace(                    \
855             max_bottom.item<float>(),        \
856             max_top.item<float>(),            \
857             num_pixels_height                \
858         ).to(torch::kFloat);
859
860
861     // converting original coordinates to their corresponding cartesian
862     float delta_r = 1/static_cast<float>(this->beamformedImage.size(0));
863     float delta_azimuth =                \
864         torch::abs(                        \
865             this->azimuth_centers.index({1}) \
866             - this->azimuth_centers.index({0}) \
867         ).item<float>();
868
869
870
871     // getting query points
872     torch::Tensor range_values = \
873         torch::linspace(                \
874             delta_r,                    \
875             this->beamformedImage.size(0) * delta_r, \
876             this->beamformedImage.size(0) \
877         ).to(torch::kFloat);
878     range_values = \
879         range_values.reshape({range_values.numel(), 1});
880     range_values = \
881         torch::tile(range_values, \
882             {1, this->azimuth_centers.numel()});
883
884
885     // getting angle-values
886     torch::Tensor angle_values =                \
887         this->azimuth_centers                    \
888         - torch::mean(this->azimuth_centers) \
889         + torch::tensor({90});
890     angle_values =                \
891         torch::tile(                \
892             angle_values,            \
893             {this->beamformedImage.size(0), 1});
894
895
896     // converting to cartesian original points
897     torch::Tensor query_original_x = \
898         range_values * torch::cos(angle_values * PI/180);
899     torch::Tensor query_original_y = \
900         range_values * torch::sin(angle_values * PI/180);
901
902
903     // converting points to vector 2D format
904     torch::Tensor query_source =                \
905         torch::cat({                            \
906             query_original_x.reshape({1, query_original_x.numel()}), \
907             query_original_y.reshape({1, query_original_y.numel()}), \
908             0;
909
910
911     // converting reflectivity to corresponding 2D format
912     torch::Tensor reflectivity_vectors = \
913         this->beamformedImage.reshape({1, this->beamformedImage.numel()});
914
915
916     // creating image
917     torch::Tensor cartesianImageLocal =                \
918         torch::zeros(                                    \
919             {num_pixels_height,

```

```

920         num_pixels_width} \
921         ).to(torch::kComplexFloat);
922
923
924     /*
925     Next Aim: start interpolating the points on the uniform grid.
926     */
927
928     #pragma omp parallel for
929     for(int x_index = 0; x_index < query_x.numel(); ++x_index){
930         // if(DEBUG_ULA) std::cout << "\t\t\t x_index = " << x_index << " ";
931         #pragma omp parallel for
932         for(int y_index = 0; y_index < query_y.numel(); ++y_index){
933             // if(DEBUG_ULA) if(y_index%16 == 0) std::cout<<".";
934
935             // getting current values
936             torch::Tensor current_x = query_x.index({x_index}).reshape({1, 1});
937             torch::Tensor current_y = query_y.index({y_index}).reshape({1, 1});
938
939
940             // getting the query value
941             torch::Tensor query_vector = torch::cat({current_x, current_y}, 0);
942
943
944             // copying the query source
945             torch::Tensor query_source_relative = query_source;
946             query_source_relative = query_source_relative - query_vector;
947
948
949             // subsetting using absolute values and masks
950             float threshold = delta_r * 10;
951             // PRINTDOTS
952             auto mask_row = \
953                 torch::abs(query_source_relative[0]) <= threshold;
954             auto mask_col = \
955                 torch::abs(query_source_relative[1]) <= threshold;
956             auto mask_together = torch::mul(mask_row, mask_col);
957
958
959
960             // calculating number of points in threshold neighbourhood
961             int num_points_in_threshold_neighbourhood = \
962                 torch::sum(mask_together).item<int>();
963             if (num_points_in_threshold_neighbourhood == 0){
964                 continue;
965             }
966
967
968             // subsetting points in neighbourhood
969             torch::Tensor PointsInNeighbourhood = \
970                 query_source_relative.index({
971                     torch::indexing::Slice(), \
972                     mask_together});
973             torch::Tensor ReflectivitiesInNeighbourhood = \
974                 reflectivity_vectors.index({torch::indexing::Slice(), mask_together});
975
976
977             // finding the distance between the points
978             torch::Tensor relativeDistances = \
979                 torch::linalg_norm(PointsInNeighbourhood, 2, 0, true, torch::kFloat);
980
981
982             // calculating weighing factor
983             torch::Tensor weighingFactor = \
984                 torch::nn::functional::softmax( \
985                     torch::max(relativeDistances)- relativeDistances, \
986                     torch::nn::functional::SoftmaxFuncOptions(1));
987
988
989             // combining intensities using distances
990             torch::Tensor finalIntensity = \
991                 torch::sum(
992                     torch::mul(weighingFactor, \

```

```

993         ReflectivitiesInNeighbourhood));
994
995         // assigning values
996         cartesianImageLocal.index_put_({x_index, y_index}, finalIntensity);
997     }
998     // std::cout<<std::endl;
999 }
1000
1001 // saving to member function
1002 this->cartesianImage = cartesianImageLocal;
1003
1004 }
1005
1006 /* =====
1007 Aim: create acoustic image directly
1008 ----- */
1009 void nfdc_createAcousticImage(ScattererClass* scatterers, \
1010                             TransmitterClass* transmitterObj){
1011
1012     // first we ensure that the scatterers are in our frame of reference
1013     scatterers->coordinates = scatterers->coordinates - this->location;
1014
1015     // finding the spherical coordinates of the scatterers
1016     torch::Tensor scatterers_spherical = fCart2Sph(scatterers->coordinates);
1017
1018     // note that its not precisely projection. its rotation. So the original lengths must be maintained.
1019     // but thats easy since the operation of putting the elevation to be zero works just fine.
1020     scatterers_spherical.index_put_({1, torch::indexing::Slice(), 0}, 0);
1021
1022     // converting the points back to cartesian
1023     torch::Tensor scatterers_acoustic_cartesian = fSph2Cart(scatterers_spherical);
1024
1025     // removing the z-dimension
1026     scatterers_acoustic_cartesian = \
1027         scatterers_acoustic_cartesian.index({torch::indexing::Slice(0, 2, 1), \
1028                                             torch::indexing::Slice()});
1029
1030     // deciding image dimensions
1031     int num_pixels_x = 512;
1032     int num_pixels_y = 512;
1033     torch::Tensor acousticImage = \
1034         torch::zeros({num_pixels_x, \
1035                     num_pixels_y}).to(torch::kFloat);
1036
1037     // finding the max and min values
1038     torch::Tensor min_x = torch::min(scatterers_acoustic_cartesian[0]);
1039     torch::Tensor max_x = torch::max(scatterers_acoustic_cartesian[0]);
1040     torch::Tensor min_y = torch::min(scatterers_acoustic_cartesian[1]);
1041     torch::Tensor max_y = torch::max(scatterers_acoustic_cartesian[1]);
1042
1043     // creating query grids
1044     torch::Tensor query_x = torch::linspace(0, 1, num_pixels_x);
1045     torch::Tensor query_y = torch::linspace(0, 1, num_pixels_y);
1046
1047     // scaling it up to image max-point spread
1048     query_x = min_x + (max_x - min_x) * query_x;
1049     query_y = min_y + (max_y - min_y) * query_y;
1050     float delta_queryx = (query_x[1] - query_x[0]).item<float>();
1051     float delta_queryy = (query_y[1] - query_y[0]).item<float>();
1052
1053     // creating a mesh-grid
1054     auto queryMeshGrid = torch::meshgrid({query_x, query_y}, "ij");
1055     query_x = queryMeshGrid[0].reshape({1, queryMeshGrid[0].numel()});
1056     query_y = queryMeshGrid[1].reshape({1, queryMeshGrid[1].numel()});
1057     torch::Tensor queryMatrix = torch::cat({query_x, query_y}, 0);
1058
1059     // printing shapes
1060     if(DEBUG_ULA) PRINTSMALLLINE
1061     if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: query_x.shape =
1062         "<<query_x.sizes().vec()<<std::endl;
1063     if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: query_y.shape =
1064         "<<query_y.sizes().vec()<<std::endl;

```

```

1063 if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: queryMatrix.shape =
      "<<queryMatrix.sizes().vec()<<std::endl;
1064
1065 // setting up threshold values
1066 float threshold_value = \
1067     std::min(delta_queryx, \
1068         delta_queryy); if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line
      711"<<std::endl;
1069
1070 // putting a loop through the whole thing
1071 for(int i = 0; i<queryMatrix[0].numel(); ++i){
1072     // for each element in the query matrix
1073     if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 716"<<std::endl;
1074
1075     // calculating relative position of all the points
1076     torch::Tensor relativeCoordinates = \
1077         scatterers_acoustic_cartesian - \
1078         queryMatrix.index({torch::indexing::Slice(), i}).reshape({2, 1}); if(DEBUG_ULA)
      std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 720"<<std::endl;
1079
1080     // calculating distances between all the points and the query point
1081     torch::Tensor relativeDistances = \
1082         torch::linalg_norm(relativeCoordinates, \
1083             1, 0, true, \
1084             torch::kFloat);if(DEBUG_ULA) std::cout<<"\t\t\t
      ULAClass::nfdc_createAcousticImage: line 727"<<std::endl;
1085
1086     // finding points that are within the threshold
1087     torch::Tensor conditionMeetingPoints = \
      relativeDistances.squeeze() <= threshold_value;if(DEBUG_ULA) std::cout<<"\t\t\t
      ULAClass::nfdc_createAcousticImage: line 729"<<std::endl;
1088
1089     // subsetting the points in the neighbourhood
1090     if(torch::sum(conditionMeetingPoints).item<float>() == 0){
1091
1092         // continuing implementation if there are no points in the neighbourhood
1093         continue; if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line
      735"<<std::endl;
1094     }
1095     else{
1096         // creating mask for points in the neighbourhood
1097         auto mask = (conditionMeetingPoints == 1);if(DEBUG_ULA) std::cout<<"\t\t\t
      ULAClass::nfdc_createAcousticImage: line 739"<<std::endl;
1098
1099         // subsetting relative distances in the neighbourhood
1100         torch::Tensor distanceInTheNeighbourhood = \
1101             relativeDistances.index({torch::indexing::Slice(), mask});if(DEBUG_ULA) std::cout<<"\t\t\t
      ULAClass::nfdc_createAcousticImage: line 743"<<std::endl;
1102
1103         // subsetting reflectivity of points in the neighbourhood
1104         torch::Tensor reflectivityInTheNeighbourhood = \
1105             scatterers->reflectivity.index({torch::indexing::Slice(), mask});if(DEBUG_ULA)
      std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 747"<<std::endl;
1106
1107         // assigning intensity as a function of distance and reflectivity
1108         torch::Tensor reflectivityAssignment = \
1109             torch::mul(torch::exp(-distanceInTheNeighbourhood), \
1110                 reflectivityInTheNeighbourhood);if(DEBUG_ULA) std::cout<<"\t\t\t
      ULAClass::nfdc_createAcousticImage: line 752"<<std::endl;
1111         reflectivityAssignment = \
1112             torch::sum(reflectivityAssignment);if(DEBUG_ULA) std::cout<<"\t\t\t
      ULAClass::nfdc_createAcousticImage: line 754"<<std::endl;
1113
1114         // assigning this value to the image pixel intensity
1115         int pixel_position_x = i%num_pixels_x;
1116         int pixel_position_y = std::floor(i/num_pixels_x);
1117         acousticImage.index_put_({pixel_position_x, \
1118             pixel_position_y}, \
1119             reflectivityAssignment.item<float>());if(DEBUG_ULA) std::cout<<"\t\t\t
      ULAClass::nfdc_createAcousticImage: line 761"<<std::endl;
1120     }
1121 }
1122
1123

```

```
1124 // storing the acoustic-image to the member
1125 this->currentArtificialAcousticImage = acousticImage;
1126
1127 // // saving the torch::tensor
1128 // torch::save(acousticImage, \
1129 //             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/Assets/acoustic_image.pt");
1130
1131
1132
1133 // // bringing it back to the original coordinates
1134 // scatterers->coordinates = scatterers->coordinates + this->location;
1135 }
```

```
1183 };
```

```

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

```

```

126 // call sync-component attributes
127 this->syncComponentAttributes();
128
129 // initializing all the ULAs
130 this->ULA_fls.init( &this->transmitter_fls);
131 this->ULA_port.init( &this->transmitter_port);
132 this->ULA_starboard.init( &this->transmitter_starboard);
133
134 // precomputing delay-matrices for the ULA-class
135 std::thread ULA_fls_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
136                                         &this->ULA_fls, \
137                                         &this->transmitter_fls);
138 std::thread ULA_port_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
139                                         &this->ULA_port, \
140                                         &this->transmitter_port);
141 std::thread ULA_starboard_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
142                                         &this->ULA_starboard, \
143                                         &this->transmitter_starboard);
144
145 // joining the threads back
146 ULA_fls_precompute_weights_t.join();
147 ULA_port_precompute_weights_t.join();
148 ULA_starboard_precompute_weights_t.join();
149
150 }
151
152
153
154 /*=====
155 Aim: stepping motion
156 -----*/
157 void step(float timestep){
158
159     // updating location
160     this->location = this->location + this->velocity * timestep;
161     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 81 \n";
162
163     // updating attributes of members
164     this->syncComponentAttributes();
165     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 85 \n";
166 }
167
168
169
170 /*=====
171 Aim: updateAttributes
172 -----*/
173 void syncComponentAttributes(){
174
175     // updating ULA attributes
176     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 97 \n";
177
178     // updating locations
179     this->ULA_fls.location = this->location;
180     this->ULA_port.location = this->location;
181     this->ULA_starboard.location = this->location;
182
183     // updating the pointing direction of the ULAs
184     torch::Tensor ula_fls_sensor_direction_spherical = \
185         fCart2Sph(this->pointing_direction); // spherical coords
186     ula_fls_sensor_direction_spherical[0] = \
187         ula_fls_sensor_direction_spherical[0] - 90;
188     torch::Tensor ula_fls_sensor_direction_cart = \
189         fSph2Cart(ula_fls_sensor_direction_spherical);
190
191     this->ULA_fls.sensorDirection = ula_fls_sensor_direction_cart; // assigning sensor direction for
192                                ULA-FLS
193     this->ULA_port.sensorDirection = -this->pointing_direction; // assigning sensor direction for
194                                ULA-Port
195     this->ULA_starboard.sensorDirection = -this->pointing_direction; // assigning sensor direction for
196                                ULA-Starboard
197
198     // // calling the function to update the arguments

```

```

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

```

```

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

```

```

333 // asking ULAs to simulate signal through multithreading
334 std::thread ulafls_signalsim_t(&ULAClass::nfdc_simulateSignal, \
335                               &this->ULA_fls, \
336                               &scatterer_fls, \
337                               &this->transmitter_fls);
338 std::thread ulaport_signalsim_t(&ULAClass::nfdc_simulateSignal, \
339                                &this->ULA_port, \
340                                &scatterer_port, \
341                                &this->transmitter_port);
342 std::thread ulastarboard_signalsim_t(&ULAClass::nfdc_simulateSignal, \
343                                     &this->ULA_starboard, \
344                                     &scatterer_starboard, \
345                                     &this->transmitter_starboard);
346
347 // joining them back
348 ulafls_signalsim_t.join(); // joining back the thread for ULA-FLS
349 ulaport_signalsim_t.join(); // joining back the signals-sim thread for ULA-Port
350 ulastarboard_signalsim_t.join(); // joining back the signal-sim thread for ULA-Starboard
351 savetensor_t.join(); // joining back the signal-sim thread for tensor-saving
352
353 }
354
355 // Imaging Function
356 /* =====
357 ----- */
358 void image(){
359
360 // asking ULAs to decimate the signals obtained at each time step
361 std::thread ULA_fls_image_t(&ULAClass::nfdc_decimateSignal, \
362                             &this->ULA_fls, \
363                             &this->transmitter_fls);
364 std::thread ULA_port_image_t(&ULAClass::nfdc_decimateSignal, \
365                              &this->ULA_port, \
366                              &this->transmitter_port);
367 std::thread ULA_starboard_image_t(&ULAClass::nfdc_decimateSignal, \
368                                   &this->ULA_starboard, \
369                                   &this->transmitter_starboard);
370
371 // joining the threads back
372 ULA_fls_image_t.join();
373 ULA_port_image_t.join();
374 ULA_starboard_image_t.join();
375
376 // saving the decimated signal
377 if (SAVE_DECIMATED_SIGNAL_MATRIX) {
378     std::cout << "\t AUVClass::image: saving decimated signal matrix" \
379                 << std::endl;
380     torch::save(this->ULA_fls.signalMatrix, \
381                "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/decimated_signalMatrix_fls.pt");
382     torch::save(this->ULA_port.signalMatrix, \
383                "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/decimated_signalMatrix_port.pt");
384     torch::save(this->ULA_starboard.signalMatrix, \
385                "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/decimated_signalMatrix_starboard.pt");
386 }
387
388 // asking ULAs to match-filter the signals
389 std::thread ULA_fls_matchfilter_t( \
390     &ULAClass::nfdc_matchFilterDecimatedSignal, \
391     &this->ULA_fls);
392 std::thread ULA_port_matchfilter_t( \
393     &ULAClass::nfdc_matchFilterDecimatedSignal, \
394     &this->ULA_port);
395 std::thread ULA_starboard_matchfilter_t( \
396     &ULAClass::nfdc_matchFilterDecimatedSignal, \
397     &this->ULA_starboard);
398
399 // joining the threads back
400 ULA_fls_matchfilter_t.join();
401 ULA_port_matchfilter_t.join();
402 ULA_starboard_matchfilter_t.join();
403
404 // saving the decimated signal

```

```

406 if (SAVE_MATCHFILTERED_SIGNAL_MATRIX) {
407
408     // saving the tensors
409     std::cout << "\t AUVClass::image: saving match-filtered signal matrix" \
410         << std::endl;
411     torch::save(this->ULA_fls.signalMatrix, \
412         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/matchfiltered_signalMatrix_fls.pt");
413     torch::save(this->ULA_port.signalMatrix, \
414         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/matchfiltered_signalMatrix_port.pt");
415     torch::save(this->ULA_starboard.signalMatrix, \
416         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/matchfiltered_signalMatrix_starboard.pt");
417
418     // running python-script
419
420 }
421
422
423
424 // performing the beamforming
425 std::thread ULA_fls_beamforming_t(&ULAClass::nfdc_beamforming, \
426     &this->ULA_fls, \
427     &this->transmitter_fls);
428 std::thread ULA_port_beamforming_t(&ULAClass::nfdc_beamforming, \
429     &this->ULA_port, \
430     &this->transmitter_port);
431 std::thread ULA_starboard_beamforming_t(&ULAClass::nfdc_beamforming, \
432     &this->ULA_starboard, \
433     &this->transmitter_starboard);
434
435 // joining the filters back
436 ULA_fls_beamforming_t.join();
437 ULA_port_beamforming_t.join();
438 ULA_starboard_beamforming_t.join();
439
440 }
441
442
443
444
445 /* =====
446 Aim: directly create acoustic image
447 ----- */
448 void createAcousticImage(ScattererClass* scatterers){
449
450     // making three copies
451     ScattererClass scatterer_fls = scatterers;
452     ScattererClass scatterer_port = scatterers;
453     ScattererClass scatterer_starboard = scatterers;
454
455     // printing size of scatterers before subsetting
456     PRINTSMALLLINE
457     std::cout << "\t > AUVClass::createAcousticImage: Beginning Scatterer Subsetting" << std::endl;
458     std::cout << "\t AUVClass::createAcousticImage: scatterer_fls.coordinates.shape (before) = ";
459         fPrintTensorSize(scatterer_fls.coordinates);
460     std::cout << "\t AUVClass::createAcousticImage: scatterer_port.coordinates.shape (before) = ";
461         fPrintTensorSize(scatterer_port.coordinates);
462     std::cout << "\t AUVClass::createAcousticImage: scatterer_starboard.coordinates.shape (before) = ";
463         fPrintTensorSize(scatterer_starboard.coordinates);
464
465     // finding the pointing direction in spherical
466     torch::Tensor auv_pointing_direction_spherical = fCart2Sph(this->pointing_direction);
467
468     // asking the transmitters to subset the scatterers by multithreading
469     std::thread transmitterFLSSubset_t(&AUVClass::subsetScatterers, this, \
470         &scatterer_fls, \
471         &this->transmitter_fls, \
472         (float)0);
473     std::thread transmitterPortSubset_t(&AUVClass::subsetScatterers, this, \
474         &scatterer_port, \
475         &this->transmitter_port, \
476         auv_pointing_direction_spherical[1].item<float>());
477     std::thread transmitterStarboardSubset_t(&AUVClass::subsetScatterers, this, \
478         &scatterer_starboard, \

```

```

476                                     &this->transmitter_starboard, \
477                                     - auv_pointing_direction_spherical[1].item<float>());
478
479     // joining the subset threads back
480     transmitterFLSSubset_t.join( );
481     transmitterPortSubset_t.join( );
482     transmitterStarboardSubset_t.join( );
483
484
485     // asking the ULAs to directly create acoustic images
486     std::thread ULA_fls_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, this->ULA_fls, \
487                                         &scatterer_fls, &this->transmitter_fls);
488     std::thread ULA_port_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, &this->ULA_port, \
489                                         &scatterer_port, &this->transmitter_port);
490     std::thread ULA_starboard_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, &this->ULA_starboard, \
491                                              &scatterer_starboard, &this->transmitter_starboard);
492
493     // joining the threads back
494     ULA_fls_acoustic_image_t.join( );
495     ULA_port_acoustic_image_t.join( );
496     ULA_starboard_acoustic_image_t.join();
497
498 }
499
500 };
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524 // 0.0000,
525 // 0.0000,
526 // 0.0000,
527 // 0.0000,
528 // 0.0000,
529 // 0.0000,
530 // 0.0000,
531 // 0.0000,
532 // 0.0000,
533 // 0.0000,
534 // 0.0000,
535 // 0.0000,
536 // 0.0000,
537 // 0.0000,
538 // 0.0000,
539 // 0.0000,
540 // 0.0000,
541 // 0.0000,
542 // 0.0000,
543 // 0.0000,
544 // 0.0000,
545 // 0.0000,
546 // 0.0000,
547 // 0.0000,
548 // 0.0000,

```

```
549 // 0.0000,  
550 // 0.0000,  
551 // 0.0000,  
552 // 0.0000,  
553 // 0.0000,  
554 // 0.0000,  
555 // 0.0001,  
556 // 0.0001,  
557 // 0.0002,  
558 // 0.0003,  
559 // 0.0006,  
560 // 0.0009,  
561 // 0.0014,  
562 // 0.0022, 0.0032, 0.0047, 0.0066, 0.0092, 0.0126, 0.0168, 0.0219, 0.0281, 0.0352, 0.0432, 0.0518, 0.0609,  
    0.0700, 0.0786, 0.0861, 0.0921, 0.0958, 0.0969, 0.0950, 0.0903, 0.0833, 0.0755, 0.0694, 0.0693, 0.0825,  
    0.1206
```

8.2 Setup Scripts

8.2.1 Seafloor Setup

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

```

1  /* =====
2  Aim: Setup sea floor
3  =====*/
4
5  // including headerfiles
6  #include <torch/torch.h>
7  #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/ScattererClass.h"
8
9  // including functions
10 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fCreateHills.cpp"
11 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fCreateBoxes.cpp"
12
13 #ifndef DEVICE
14     #define DEVICE          torch::kCPU
15     // #define DEVICE      torch::kMPS
16     // #define DEVICE      torch::kCUDA
17 #endif
18
19 // adding terrrain features
20 #define BOXES                false
21 #define HILLS                true
22 #define DEBUG_SEAFLOOR      false
23 #define SAVETENSORS_Seafloor false
24 #define PLOT_SEAFLOOR       false
25
26 // functin that setups the sea-floor
27 void SeafloorSetup(ScattererClass* scatterers) {
28
29     // sea-floor bounds
30     int bed_width = 100; // width of the bed (x-dimension)
31     int bed_length = 100; // length of the bed (y-dimension)
32
33     // creating some tensors to pass. This is put outside to maintain scope
34     torch::Tensor box_coordinates = torch::zeros({3,1}).to(torch::kFloat).to(DEVICE);
35     torch::Tensor box_reflectivity = torch::zeros({1,1}).to(torch::kFloat).to(DEVICE);
36
37     // creating boxes
38     if (BOXES)
39         fCreateBoxes(bed_width, \
40                     bed_length, \
41                     box_coordinates, \
42                     box_reflectivity);
43
44     // scatter-intensity
45     // int bed_width_density = 100; // density of points along x-dimension
46     // int bed_length_density = 100; // density of points along y-dimension
47     int bed_width_density = 10; // density of points along x-dimension
48     int bed_length_density = 10; // density of points along y-dimension
49
50     // setting up coordinates
51     auto xpoints = torch::linspace(0, \
52                                   bed_width, \
53                                   bed_width * bed_width_density).to(DEVICE);
54     auto ypoints = torch::linspace(0, \
55                                   bed_length, \
56                                   bed_length * bed_length_density).to(DEVICE);
57
58     // creating mesh
59     auto mesh_grid = torch::meshgrid({xpoints, ypoints}, "ij");
60     auto X = mesh_grid[0];
61     auto Y = mesh_grid[1];
62     X = torch::reshape(X, {1, X.numel()});
63     Y = torch::reshape(Y, {1, Y.numel()});
64
65     // creating heights of scattereres

```

```

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

```

```
137         std::cout<<"SeafloorSetup: Saved Seafloor "<<std::endl;
138     }
139
140 }
```

8.2.2 Transmitter Setup

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

```

1  /* =====
2  Aim: Setup sea floor
3  =====*/
4  #include <torch/torch.h>
5  #include <cmath>
6
7  #ifndef DEVICE
8      // #define DEVICE      torch::kMPS
9      #define DEVICE      torch::kCPU
10 #endif
11
12
13
14 // function to calibrate the transmitters
15 void TransmitterSetup(TransmitterClass* transmitter_fls,
16                       TransmitterClass* transmitter_port,
17                       TransmitterClass* transmitter_starboard) {
18
19     // Setting up transmitter
20     float sampling_frequency = 160e3;           // sampling frequency
21     float f1                 = 50e3;           // first frequency of LFM
22     float f2                 = 70e3;           // second frequency of LFM
23     float fc                 = (f1 + f2)/2;     // finding center-frequency
24     float bandwidth          = std::abs(f2 - f1); // bandwidth
25     float pulselength        = 5e-2;           // time of recording
26
27     // building LFM
28     torch::Tensor timearray = torch::linspace(0, \
29                                              pulselength, \
30                                              floor(pulselength * sampling_frequency)).to(DEVICE);
31     float K                 = (f2 - f1)/pulselength; // calculating frequency-slope
32     torch::Tensor Signal    = K * timearray;         // frequency at each time-step, with f1 = 0
33     Signal                  = torch::mul(2*PI*(f1 + Signal), \
34                                       timearray);    // creating
35     Signal                  = cos(Signal);           // calculating signal
36
37
38     // Setting up transmitter
39     torch::Tensor location   = torch::zeros({3,1}).to(DEVICE); // location of transmitter
40     float azimuthal_angle_fls = 0;                       // initial pointing direction
41     float azimuthal_angle_port = 90;                     // initial pointing direction
42     float azimuthal_angle_starboard = -90;                // initial pointing direction
43
44     float elevation_angle    = -60;                       // initial pointing direction
45
46     float azimuthal_beamwidth_fls = 20;                   // azimuthal beamwidth of the signal cone
47     float azimuthal_beamwidth_port = 20;                 // azimuthal beamwidth of the signal cone
48     float azimuthal_beamwidth_starboard = 20;            // azimuthal beamwidth of the signal cone
49
50     float elevation_beamwidth_fls = 20;                   // elevation beamwidth of the signal cone
51     float elevation_beamwidth_port = 20;                 // elevation beamwidth of the signal cone
52     float elevation_beamwidth_starboard = 20;            // elevation beamwidth of the signal cone
53
54     int azimuthQuantDensity      = 10; // number of points, a degree is split into quantization density
55                                     // along azimuth (used for shadowing)
56     int elevationQuantDensity    = 10; // number of points, a degree is split into quantization density
57                                     // along elevation (used for shadowing)
58     float rangeQuantSize        = 10; // the length of a cell (used for shadowing)
59
60     float azimuthShadowThreshold = 1; // azimuth threshold (in degrees)
61     float elevationShadowThreshold = 1; // elevation threshold (in degrees)
62
63     // transmitter-fls
64     transmitter_fls->location      = location;           // Assigning location
65     transmitter_fls->Signal        = Signal;             // Assigning signal
66     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.

```

1  /* =====
2  Aim: Setup sea floor
3  NOAA: 50 to 100 KHz is the transmission frequency
4  we'll create our LFM with 50 to 70KHz
5  =====*/
6
7
8  // Choosing device
9  #ifndef DEVICE
10     // #define DEVICE      torch::kMPS
11     #define DEVICE      torch::kCPU
12 #endif
13
14
15 // the coefficients for the low-pass filter.
16 #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
17
18
19
20
21 void ULASetup(ULAClass* ula_fls,
22               ULAClass* ula_port,
23               ULAClass* ula_starboard) {
24
25     // setting up ula
26     int num_sensors      = 64;                // number of sensors
27     float sampling_frequency = 160e3;          // sampling frequency
28     float inter_element_spacing = 1500/(2*sampling_frequency); // space between samples
29     float recording_period   = 10e-2;          // sampling-period
30
31     // building the direction for the sensors
32     torch::Tensor ULA_direction = torch::tensor({-1,0,0}).reshape({3,1}).to(torch::kFloat).to(DEVICE);
33     ULA_direction               = ULA_direction/torch::linalg_norm(ULA_direction, 2, 0, true,
        torch::kFloat).to(DEVICE);
34     ULA_direction               = ULA_direction * inter_element_spacing;
35
36     // building the coordinates for the sensors
37     torch::Tensor ULA_coordinates = torch::mul(torch::linspace(0, num_sensors-1, num_sensors).to(DEVICE), \
        ULA_direction);
38
39
40     // the coefficients for the decimation filter
41     torch::Tensor lowpassfiltercoefficients =
        torch::tensor({LOWPASS_DECIMATE_FILTER_COEFFICIENTS}).to(torch::kFloat);
42
43     // assigning values
44     ula_fls->num_sensors      = num_sensors;    // assigning number of sensors
45     ula_fls->inter_element_spacing = inter_element_spacing; // assigning inter-element spacing
46     ula_fls->coordinates      = ULA_coordinates; // assigning ULA coordinates
47     ula_fls->sampling_frequency = sampling_frequency; // assigning sampling frequencys
48     ula_fls->recording_period  = recording_period; // assigning recording period
49     ula_fls->sensorDirection   = ULA_direction; // ULA direction
50     ula_fls->lowpassFilterCoefficientsForDecimation = lowpassfiltercoefficients;
51
52     // assigning values
53     ula_port->num_sensors      = num_sensors;    // assigning number of sensors
54     ula_port->inter_element_spacing = inter_element_spacing; // assigning inter-element spacing
55     ula_port->coordinates      = ULA_coordinates; // assigning ULA coordinates
56     ula_port->sampling_frequency = sampling_frequency; // assigning sampling frequencys
57     ula_port->recording_period  = recording_period; // assigning recording period
58     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.

```

1  /* =====
2  Aim: Setup sea floor
3  NOAA: 50 to 100 KHz is the transmission frequency
4  we'll create our LFM with 50 to 70KHz
5  =====*/
6
7  #ifndef DEVICE
8      #define DEVICE      torch::kMPS
9      // #define DEVICE    torch::kCPU
10 #endif
11
12 // =====
13 void AUVSetup(AUVClass* auv) {
14
15     // building properties for the auv
16     torch::Tensor location      = torch::tensor({0,50,30}).reshape({3,1}).to(torch::kFloat).to(DEVICE); //
17     // starting location of AUV
18     torch::Tensor velocity      = torch::tensor({5,0, 0}).reshape({3,1}).to(torch::kFloat).to(DEVICE); //
19     // starting velocity of AUV
20     torch::Tensor pointing_direction = torch::tensor({1,0, 0}).reshape({3,1}).to(torch::kFloat).to(DEVICE);
21     // pointing direction of AUV
22
23     // assigning
24     auv->location      = location;          // assigning location of auv
25     auv->velocity       = velocity;          // assigning vector representing velocity
26     auv->pointing_direction = pointing_direction; // assigning pointing direction of auv
27 }

```

8.3 Function Definitions

8.3.1 Cartesian Coordinates to Spherical Coordinates

```

1  /* =====
2  Aim: Setup sea floor
3  =====*/
4  #include <torch/torch.h>
5  #include <iostream>
6
7  // hash-defines
8  #define PI 3.14159265
9  #define DEBUG_Cart2Sph false
10
11 #ifndef DEVICE
12     #define DEVICE torch::kMPS
13     // #define DEVICE torch::kCPU
14 #endif
15
16
17 // bringing in functions
18 #include "/Users/vrsreeganesht/Documents/GitHub/AUV/Code/C++/Functions/fPrintTensorSize.cpp"
19
20 #pragma once
21
22 torch::Tensor fCart2Sph(torch::Tensor cartesian_vector){
23
24     // sending argument to the device
25     cartesian_vector = cartesian_vector.to(DEVICE);
26     if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 26 \n";
27
28     // splatting the point onto xy plane
29     torch::Tensor xysplat = cartesian_vector.clone().to(DEVICE);
30     xysplat[2] = 0;
31     if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 31 \n";
32
33     // finding splat lengths
34     torch::Tensor xysplat_lengths = torch::linalg_norm(xysplat, 2, 0, true, torch::kFloat).to(DEVICE);
35     if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 35 \n";
36
37     // finding azimuthal and elevation angles
38     torch::Tensor azimuthal_angles = torch::atan2(xysplat[1], xysplat[0]).to(DEVICE) * 180/PI;
39     azimuthal_angles = azimuthal_angles.reshape({1, azimuthal_angles.numel()});
40     torch::Tensor elevation_angles = torch::atan2(cartesian_vector[2], xysplat_lengths).to(DEVICE) * 180/PI;
41     torch::Tensor rho_values = torch::linalg_norm(cartesian_vector, 2, 0, true, torch::kFloat).to(DEVICE);
42     if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 42 \n";
43
44
45     // printing values for debugging
46     if (DEBUG_Cart2Sph){
47         std::cout<<"azimuthal_angles.shape = "; fPrintTensorSize(azimuthal_angles);
48         std::cout<<"elevation_angles.shape = "; fPrintTensorSize(elevation_angles);
49         std::cout<<"rho_values.shape = "; fPrintTensorSize(rho_values);
50     }
51     if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 51 \n";
52
53     // creating tensor to send back
54     torch::Tensor spherical_vector = torch::cat({azimuthal_angles, \
55                                                 elevation_angles, \
56                                                 rho_values}, 0).to(DEVICE);
57     if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 57 \n";
58
59     // returning the value
60     return spherical_vector;
61 }

```

8.3.2 Spherical Coordinates to Cartesian Coordinates

```

1  /* =====
2  Aim: Setup sea floor
3  =====*/
4  #include <torch/torch.h>
5
6  #pragma once
7
8  // hash-defines
9  #define PI          3.14159265
10 #define MYDEBUGFLAG false
11
12 #ifndef DEVICE
13     // #define DEVICE      torch::kMPS
14     #define DEVICE      torch::kCPU
15 #endif
16
17
18 torch::Tensor fSph2Cart(torch::Tensor spherical_vector){
19
20
21
22     // sending argument to device
23     spherical_vector = spherical_vector.to(DEVICE);
24
25     // creating cartesian vector
26     torch::Tensor cartesian_vector =
27         torch::zeros({3,(int)(spherical_vector.numel()/3)}).to(torch::kFloat).to(DEVICE);
28
29     // populating it
30     cartesian_vector[0] = spherical_vector[2] * \
31         torch::cos(spherical_vector[1] * PI/180) * \
32         torch::cos(spherical_vector[0] * PI/180);
33     cartesian_vector[1] = spherical_vector[2] * \
34         torch::cos(spherical_vector[1] * PI/180) * \
35         torch::sin(spherical_vector[0] * PI/180);
36     cartesian_vector[2] = spherical_vector[2] * \
37         torch::sin(spherical_vector[1] * PI/180);
38
39     // returning the value
40     return cartesian_vector;
41 }

```

8.3.3 Column-Wise Convolution

```

1  /* =====
2  Aim: Convolve the columns of two input matrices
3  =====*/
4  #include <ratio>
5  #include <stdexcept>
6  #include <torch/torch.h>
7
8  #pragma once
9
10 // hash-defines
11 #define PI          3.14159265
12 #define MYDEBUGFLAG false
13
14 #ifndef DEVICE
15     // #define DEVICE      torch::kMPS
16     #define DEVICE      torch::kCPU
17 #endif
18
19
20 void fConvolveColumns(torch::Tensor& inputMatrix, \
21     torch::Tensor& kernelMatrix){

```

```

22
23
24 // printing shape
25 if(MYDEBUGFLAG) std::cout<<"inputMatrix.shape =
    [<<inputMatrix.size(0)<<","<<inputMatrix.size(1)<<std::endl;
26 if(MYDEBUGFLAG) std::cout<<"kernelMatrix.shape =
    [<<kernelMatrix.size(0)<<","<<kernelMatrix.size(1)<<std::endl;
27
28 // ensuring the two have the same number of columns
29 if (inputMatrix.size(1) != kernelMatrix.size(1)){
30     throw std::runtime_error("fConvolveColumns: arguments cannot have different number of columns");
31 }
32
33
34 // calculating length of final result
35 int final_length = inputMatrix.size(0) + kernelMatrix.size(0) - 1; if(MYDEBUGFLAG) std::cout<<"\t\t\t
    fConvolveColumns: 27"<<std::endl;
36
37 // calculating FFT of the two matrices
38 torch::Tensor inputMatrix_FFT = torch::fft::fftn(inputMatrix, \
39     {final_length}, \
40     {0}); if(MYDEBUGFLAG) std::cout<<"\t\t\t fConvolveColumns:
        32"<<std::endl;
41 torch::Tensor kernelMatrix_FFT = torch::fft::fftn(kernelMatrix, \
42     {final_length}, \
43     {0}); if(MYDEBUGFLAG) std::cout<<"\t\t\t fConvolveColumns:
        35"<<std::endl;
44
45 // element-wise multiplying the two matrices
46 torch::Tensor MulProduct = torch::mul(inputMatrix_FFT, kernelMatrix_FFT); if(MYDEBUGFLAG)
    std::cout<<"\t\t\t fConvolveColumns: 38"<<std::endl;
47
48 // finding the inverse FFT
49 torch::Tensor convolvedResult = torch::fft::ifftn(MulProduct, \
50     {MulProduct.size(0)}, \
51     {0}); if(MYDEBUGFLAG) std::cout<<"\t\t\t fConvolveColumns:
        43"<<std::endl;
52
53 // over-riding the result with the input so that we can save memory
54 inputMatrix = convolvedResult; if(MYDEBUGFLAG) std::cout<<"\t\t\t fConvolveColumns: 46"<<std::endl;
55
56 }

```

8.3.4 Buffer 2D

```

1 /* =====
2 Aim: Convolver the columns of two input matrices
3 =====*/
4 #include <stdexcept>
5 #include <torch/torch.h>
6
7 #pragma once
8
9 // hash-defines
10 #ifndef DEVICE
11     // #define DEVICE      torch::kMPS
12     #define DEVICE      torch::kCPU
13 #endif
14
15 // #define DEBUG_Buffer2D true
16 #define DEBUG_Buffer2D false
17
18
19 void fBuffer2D(torch::Tensor& inputMatrix,
20     int frame_size){
21
22     // ensuring the first dimension is 1.
23     if(inputMatrix.size(0) != 1){
24         throw std::runtime_error("fBuffer2D: The first-dimension must be 1 \n");
25     }

```

```

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

```

8.3.5 fAnglesToTensor

```

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

```

8.3.6 fCalculateCosine

```
1 // including headerfiles
2 #include <torch/torch.h>
3
4 // function to calculate cosine of two tensors
5 torch::Tensor fCalculateCosine(torch::Tensor inputTensor1,
6                               torch::Tensor inputTensor2)
7 {
8     // column normalizing the the two signals
9     inputTensor1 = fColumnNormalize(inputTensor1);
10    inputTensor2 = fColumnNormalize(inputTensor2);
11
12    // finding their dot product
13    torch::Tensor dotProduct = inputTensor1 * inputTensor2;
14    torch::Tensor cosineBetweenVectors = torch::sum(dotProduct,
15                                                    0,
16                                                    true);
17
18    // returning the value
19    return cosineBetweenVectors;
20 }
21
```

8.4 Main Scripts

8.4.1 Signal Simulation

1.

```

1  /*=====
2  Aim: Signal Simulation
3  =====*/
4
5
6  // including standard
7  #include <ostream>
8  #include <torch/torch.h>
9  #include <iostream>
10 #include <thread>
11 #include "math.h"
12 #include <chrono>
13 #include <Python.h>
14 #include <Eigen/Dense>
15 #include <cstdlib>      // For terminal access
16 #include <omp.h>       // the openMP
17
18 // hash defines
19 #ifndef PRINTSPACE
20     #define PRINTSPACE    std::cout<<"\n\n\n";
21 #endif
22 #ifndef PRINTSMALLLINE
23     #define PRINTSMALLLINE
24     std::cout<<"-----" <<std::endl;
25 #endif
26 #ifndef PRINTDOTS
27     #define PRINTDOTS
28     std::cout<<"....." <<std::endl;
29 #endif
30 #ifndef PRINTLINE
31     #define PRINTLINE
32     std::cout<<"===== " <<std::endl;
33 #endif
34 #ifndef PI
35     #define PI            3.14159265
36 #endif
37
38 // debugging hashdefine
39 #ifndef DEBUGMODE
40     #define DEBUGMODE     false
41 #endif
42
43 // deciding to save tensors or not
44 #ifndef SAVETENSORS
45     #define SAVETENSORS    true
46     // #define SAVETENSORS    false
47 #endif
48
49 // choose device here
50 #ifndef DEVICE
51     #define DEVICE        torch::kCPU
52     // #define DEVICE        torch::kMPS
53     // #define DEVICE        torch::kCUDA
54 #endif
55
56 // Enable Imaging
57 #define IMAGING_TOGGLE    true
58
59 // class definitions
60 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/ScattererClass.h"
61 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/ULAClass.h"
62 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/TransmitterClass.h"
63 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/AUVClass.h"

```

```

62
63 // setup-scripts
64 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/ULASetup/ULASetup.cpp"
65 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/TransmitterSetup/TransmitterSetup.cpp"
66 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/SeafloorSetup/SeafloorSetup.cpp"
67 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/AUVSetup/AUVSetup.cpp"
68
69 // functions
70 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fPrintTensorSize.cpp"
71 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fSph2Cart.cpp"
72 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fCart2Sph.cpp"
73 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fConvolveColumns.cpp"
74 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fRunSystemScriptInSeperateThread.cpp"
75
76
77 // main-function
78 int main() {
79
80     // Ensuring no-gradients are calculated in this scope
81     torch::NoGradGuard no_grad;
82
83     // Building Sea-floor
84     ScattererClass SeafloorScatter;
85     std::thread scatterThread_t(SeafloorSetup, \
86                                 &SeafloorScatter);
87
88     // Building ULA
89     ULAClass ula_fls, ula_port, ula_starboard;
90     std::thread ulaThread_t(ULASetup, \
91                             &ula_fls, \
92                             &ula_port, \
93                             &ula_starboard);
94
95     // Building Transmitter
96     TransmitterClass transmitter_fls, transmitter_port, transmitter_starboard;
97     std::thread transmitterThread_t(TransmitterSetup,
98                                     &transmitter_fls,
99                                     &transmitter_port,
100                                     &transmitter_starboard);
101
102     // Joining threads
103     ulaThread_t.join(); // making the ULA population thread join back
104     transmitterThread_t.join(); // making the transmitter population thread join back
105     scatterThread_t.join(); // making the scattetr population thread join back
106
107     // building AUV
108     AUVClass auv; // instantiating class object
109     AUVSetup(&auv); // populating
110
111     // attaching components to the AUV
112     auv.ULA_fls = ula_fls; // attaching ULA-FLS to AUV
113     auv.ULA_port = ula_port; // attaching ULA-Port to AUV
114     auv.ULA_starboard = ula_starboard; // attaching ULA-Starboard to AUV
115     auv.transmitter_fls = transmitter_fls; // attaching Transmitter-FLS to AUV
116     auv.transmitter_port = transmitter_port; // attaching Transmitter-Port to AUV
117     auv.transmitter_starboard = transmitter_starboard; // attaching Transmitter-Starboard to AUV
118
119     // storing
120     ScattererClass SeafloorScatter_deepcopy = SeafloorScatter;
121
122     // pre-computing the data-structures required for processing
123     auv.init();
124
125     // mimicking movement
126     int number_of_stophops = 1;
127     // if (true) return 0;
128     for(int i = 0; i<number_of_stophops; ++i){
129
130         // time measuring
131         auto start_time = std::chrono::high_resolution_clock::now();
132
133         // printing some spaces
134         PRINTSPACE; PRINTSPACE; PRINTLINE; std::cout<<"i = "<<i<<std::endl; PRINTLINE

```

```

135
136 // making the deep copy
137 ScattererClass SeafloorScatter = SeafloorScatter_deepcopy;
138
139 // signal simulation
140 auv.simulateSignal(SeafloorScatter);
141
142
143
144 // saving simulated signal
145 if (SAVETENSORS) {
146
147     // saving the signal matrix tensors
148     torch::save(auv.ULA_fls.signalMatrix, \
149         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_fls.pt");
150     torch::save(auv.ULA_port.signalMatrix, \
151         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_port.pt");
152     torch::save(auv.ULA_starboard.signalMatrix, \
153         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_starboard.pt");
154
155     // running python script
156     std::string script_to_run =
157         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/Python/Plot_SignalMatrix.py";
158     std::thread plotSignalMatrix_t(fRunSystemScriptInSeperateThread, \
159                                     script_to_run);
160     plotSignalMatrix_t.detach();
161 }
162
163
164 if (IMAGING_TOGGLE) {
165     // creating image from signals
166     auv.image();
167
168     // saving the tensors
169     if (SAVETENSORS) {
170         // saving the beamformed images
171         torch::save(auv.ULA_fls.beamformedImage, \
172             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/ULA_fls_image.pt");
173         // torch::save(auv.ULA_port.beamformedImage, \
174             // "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/ULA_port_image.pt");
175         // torch::save(auv.ULA_starboard.beamformedImage, \
176             // "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/ULA_starboard_image.pt");
177
178         // saving cartesian image
179         torch::save(auv.ULA_fls.cartesianImage, \
180             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/ULA_fls_cartesianImage.pt");
181
182         // // running python file
183         // system("python
184             // /Users/vrsreeganesh/Documents/GitHub/AUV/Code/Python/Plot_BeamformedImage.py");
185         system("python /Users/vrsreeganesh/Documents/GitHub/AUV/Code/Python/Plot_cartesianImage.py");
186     }
187 }
188
189
190 // measuring and printing time taken
191 auto end_time = std::chrono::high_resolution_clock::now();
192 std::chrono::duration<double> time_duration = end_time - start_time;
193 PRINTDOTS; std::cout<<"Time taken (i = "<<i<<" = "<<time_duration.count()<<" seconds"<<std::endl;
194     PRINTDOTS
195
196 // moving to next position
197 auv.step(0.5);
198 }
199
200
201
202
203

```



```
204
205
206 // returning
207 return 0;
208 }
```

Chapter 9

Reading

9.1 Primary Books

- 1.

9.2 Interesting Papers