

# Autonomous Underwater Vehicle: A Surveillance Protocol

S.V. Rajendran

February 15, 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

<b>Preface</b>	<b>i</b>
<b>Introduction</b>	<b>ii</b>
<b>1 Setup</b>	<b>1</b>
1.1 Overview . . . . .	1
<b>2 Underwater Environment Setup</b>	<b>2</b>
2.1 Sea “Floor” . . . . .	2
2.2 Simple Structures . . . . .	3
2.2.1 Boxes . . . . .	3
2.2.2 Sphere . . . . .	4
<b>3 Hardware Setup</b>	<b>5</b>
3.1 Transmitter . . . . .	5
3.2 Uniform Linear Array . . . . .	6
3.3 Marine Vessel . . . . .	6
<b>4 Signal Simulation</b>	<b>7</b>
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
<b>5 Imaging</b>	<b>10</b>
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
<b>6 Control Pipeline</b>	<b>14</b>
<b>7 Results</b>	<b>16</b>
<b>8 Software</b>	<b>17</b>
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
<b>9</b>	<b>Reading</b>	<b>69</b>
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,  $\phi$  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,  $\theta$  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 signa's quadratic component. All this leads to the best candidate: the complex conjugate of the transmitted signal. However, since we're now working with the quadrature demodulated signal, the matched filter is the complex conjugate of the quadrature demodulated transmitted signal.
7. So once the filter is made, convolving that with the received signal produces a number of spikes in the processed signal. Note that due to working in the digital domain and some other factors, the spikes will not be perfect. Thus it is not safe to take the `abs()` or `real()` just yet. We'll do that after beamforming.
8. But so far, this marks the first step of the perception pipeline.

---

**Algorithm 7** Match-Filtering
 

---

**ScatterCoordinates**  $\leftarrow$   
**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**

# Chapter 8

## Software

### Overview

- 

## 8.1 Class Definitions

### 8.1.1 Class: Scatter

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

---

```
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\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     // std::cout<<"\t TransmitterClass: AUV_pointing_vector_spherical =
165     // calculating azimuth and elevation of transmitter object
166     torch::Tensor absolute_azimuth_of_transmitter = yaw +
167     torch::tensor({this->azimuthal_angle}).to(torch::kFloat).to(DEVICE);
168     torch::Tensor absolute_elevation_of_transmitter = pitch +
169     torch::tensor({this->elevation_angle}).to(torch::kFloat).to(DEVICE);
170     if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 149 \n";
171
172     // std::cout<<"\t TransmitterClass: this->azimuthal_angle = "<<this->azimuthal_angle<<std::endl;
173     // std::cout<<"\t TransmitterClass: this->elevation_angle = "<<this->elevation_angle<<std::endl;
174     // std::cout<<"\t TransmitterClass: absolute_azimuth_of_transmitter =
175     // std::cout<<"\t TransmitterClass: absolute_elevation_of_transmitter =
176     // std::cout<<"\t TransmitterClass: absolute_elevation_of_transmitter =
177
178     // converting back to Cartesian
179     torch::Tensor pointing_direction_spherical = torch::zeros({3,1}).to(torch::kFloat).to(DEVICE);
180     pointing_direction_spherical[0] = absolute_azimuth_of_transmitter;
181     pointing_direction_spherical[1] = absolute_elevation_of_transmitter;
182     pointing_direction_spherical[2] = torch::tensor({1}).to(torch::kFloat).to(DEVICE);
183     if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 60 \n";
184
185     this->pointing_direction = fSph2Cart(pointing_direction_spherical);
186     if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 169 \n";
187
188 }
189
190 /*=====
191 Aim: Subsetting Scatterers inside the cone
192 -----*/
193 Note:
194     steps:
195     1. Find azimuth and range of all points.
196     2. Find azimuth and range of current pointing vector.
197     3. Subtract azimuth and range of points from that of azimuth and range of current pointing vector
198     4. Use tilted ellipse equation to find points in the ellipse
199 -----*/
200 void subsetScatterers(ScattererClass* scatterers,
201     float tilt_angle){
202
203     // translationally change origin
204     scatterers->coordinates = scatterers->coordinates - this->location; if(DEBUGMODE_TRANSMITTER)
205     std::cout<<"\t\t TransmitterClass: line 188 "<<std::endl;
206
207     /*
208     Note: I think something we can do is see if we can subset the matrices by checking coordinate values
209     right away. If one of the coordinate values is x (relative coordinates), we know for sure that
210     the distance is greater than x, for sure. So, maybe that's something that we can work with

```

```

203  */
204
205  // Finding spherical coordinates of scatterers and pointing direction
206  torch::Tensor scatterers_spherical = fCart2Sph(scatterers->coordinates);
207  if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 191 "<<std::endl;
208  torch::Tensor pointing_direction_spherical = fCart2Sph(this->pointing_direction);
209  if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 192 "<<std::endl;
210
211  // Calculating relative azimuths and radians
212  torch::Tensor relative_spherical = scatterers_spherical - pointing_direction_spherical;
213  if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 199 "<<std::endl;
214
215  // clearing some stuff up
216  scatterers_spherical.reset(); if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line
217  202 "<<std::endl;
218  pointing_direction_spherical.reset(); if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass:
219  line 203 "<<std::endl;
220
221  // tensor corresponding to switch.
222  torch::Tensor tilt_angle_Tensor = torch::tensor({tilt_angle}).to(torch::kFloat).to(DEVICE);
223  if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 206 "<<std::endl;
224
225  // calculating length of axes
226  torch::Tensor axis_a = torch::tensor({this->azimuthal_beamwidth / 2}).to(torch::kFloat).to(DEVICE);
227  if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 208 "<<std::endl;
228  torch::Tensor axis_b = torch::tensor({this->elevation_beamwidth / 2}).to(torch::kFloat).to(DEVICE);
229  if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 209 "<<std::endl;
230
231  // part of calculating the tilted ellipse
232  torch::Tensor xcosa = relative_spherical[0] * torch::cos(tilt_angle_Tensor * PI/180);
233  torch::Tensor ysina = relative_spherical[1] * torch::sin(tilt_angle_Tensor * PI/180);
234  torch::Tensor xsina = relative_spherical[0] * torch::sin(tilt_angle_Tensor * PI/180);
235  torch::Tensor ycosa = relative_spherical[1] * torch::cos(tilt_angle_Tensor * PI/180);
236  relative_spherical.reset(); if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 215
237  "<<std::endl;
238
239  // finding points inside the tilted ellipse
240  torch::Tensor scatter_boolean = torch::div(torch::square(xcosa + ysina), torch::square(axis_a)) + \
241  torch::div(torch::square(xsina - ycosa), torch::square(axis_b)) <= 1;
242  if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line
243  221 "<<std::endl;
244
245  // clearing
246  xcosa.reset(); ysina.reset(); xsina.reset(); ycosa.reset(); if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t
247  TransmitterClass: line 224 "<<std::endl;
248
249  // subsetting points within the elliptical beam
250  auto mask = (scatter_boolean == 1); // creating a mask
251  scatterers->coordinates = scatterers->coordinates.index({torch::indexing::Slice(), mask});
252  scatterers->reflectivity = scatterers->reflectivity.index({torch::indexing::Slice(), mask});
253  if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 229 "<<std::endl;
254
255  // this is where histogram shadowing comes in (later)
256  if (ENABLE_RAYTRACING) {rangeHistogramShadowing(scatterers); std::cout<<"\t\t TransmitterClass: line
257  232 "<<std::endl;}
258
259  // translating back to the points
260  scatterers->coordinates = scatterers->coordinates + this->location;
261
262  }
263
264  /*****
265  Aim: Shadowing method (range-histogram shadowing)
266  .....
267  Note:
268  > cut down the number of threads into range-cells
269  > for each range cell, calculate histogram
270  >
271  std::cout<<"\t TransmitterClass: "
272  -----*/
273  void rangeHistogramShadowing(ScattererClass* scatterers){
274
275  // converting points to spherical coordinates

```

```

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

```

```

324     TransmitterClass: line 308"<<std::endl;
325     std::cout<<"\t\t finalReflectivityBox.shape = "; fPrintTensorSize(finalReflectivityBox);
326     torch::Tensor final_reflectivity = finalReflectivityBox.reshape({finalReflectivityBox.numel()});
327     std::cout<<"\t\t TransmitterClass: line 310"<<std::endl;
328     torch::Tensor test_checkbox = checkbox.reshape({checkbox.numel()}); std::cout<<"\t\t TransmitterClass:
329     line 311"<<std::endl;
330
331     // just taking the points corresponding to the filled. Else, there's gonna be a lot of zero zero zero
332     tensors
333     auto mask = (test_checkbox == 1); std::cout<<"\t\t TransmitterClass: line 319"<<std::endl;
334     final_coords_cart = final_coords_cart.index({torch::indexing::Slice(), mask}); std::cout<<"\t\t
335     TransmitterClass: line 320"<<std::endl;
336     final_reflectivity = final_reflectivity.index({mask}); std::cout<<"\t\t TransmitterClass: line
337     321"<<std::endl;
338
339     // overwriting the scatterers
340     scatterers->coordinates = final_coords_cart;
341     scatterers->reflectivity = final_reflectivity; std::cout<<"\t\t TransmitterClass: line 324"<<std::endl;
342 }
343
344 void internal_subsetCurrentRangeCell(torch::Tensor rangeupperlimit, \
345                                     ScattererClass* scatterers, \
346                                     torch::Tensor& checkbox, \
347                                     torch::Tensor& finalScatterBox, \
348                                     torch::Tensor& finalReflectivityBox, \
349                                     torch::Tensor& azimuth_centers, \
350                                     torch::Tensor& elevation_centers, \
351                                     torch::Tensor& spherical_coordinates){
352
353     // finding indices for points in the current range-cell
354     torch::Tensor pointsincurrentrangeCell = \
355         torch::mul((spherical_coordinates[2] <= rangeupperlimit) , \
356                   (spherical_coordinates[2] > rangeupperlimit - this->rangeQuantSize));
357
358     // checking out if there are no points in this range-cell
359     int num311 = torch::sum(pointsincurrentrangeCell).item<int>();
360     if(num311 == 0) return;
361
362     // calculating delta values
363     float delta_azimuth = azimuth_centers[1].item<float>() - azimuth_centers[0].item<float>();
364     float delta_elevation = elevation_centers[1].item<float>() - elevation_centers[0].item<float>();
365
366     // subsetting points in the current range-cell
367     auto mask = (pointsincurrentrangeCell == 1); // creating a mask
368     torch::Tensor reflectivityincurrentrangeCell =
369         scatterers->reflectivity.index({torch::indexing::Slice(), mask});
370     pointsincurrentrangeCell = spherical_coordinates.index({torch::indexing::Slice(),
371     mask});
372
373     // finding number of azimuth sizes and what not
374     int numazimuthcells = azimuth_centers.numel();
375     int numelevationcells = elevation_centers.numel();
376
377     // go through all the combinations
378     for(int azi_index = 0; azi_index < numazimuthcells; ++azi_index){
379         for(int ele_index = 0; ele_index < numelevationcells; ++ele_index){
380
381             // check if this particular azimuth-elevation direction has been taken-care of.
382             if (checkbox[ele_index][azi_index].item<bool>()) break;
383
384             // init (verified)
385             torch::Tensor current_azimuth = azimuth_centers.index({azi_index});
386             torch::Tensor current_elevation = elevation_centers.index({ele_index});
387
388             // // finding azimuth boolean
389             // torch::Tensor azi_neighbours = torch::abs(pointsincurrentrangeCell[0] - current_azimuth);
390             // azi_neighbours = azi_neighbours <= delta_azimuth; // tinker with this.
391
392             // // finding elevation boolean
393             // torch::Tensor ele_neighbours = torch::abs(pointsincurrentrangeCell[1] - current_elevation);
394             // ele_neighbours = ele_neighbours <= delta_elevation;

```

```

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

```

---

### 8.1.3 Class: Uniform Linear Array

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

---

```

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

```



```

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

```



```

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

```

```

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

```

```

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

```

```

351         //                               uniqueCounts.to(torch::kFloat));
352
353     // }
354
355     // Creating matrix out of transmitted signal
356     torch::Tensor signalTensorAsArgument = \
357         transmitterObj->Signal.reshape({transmitterObj->Signal.numel(),1});
358     signalTensorAsArgument = torch::tile(signalTensorAsArgument, \
359         {1, this->signalMatrix.size(1)});
360
361     // convolving the pulse-matrix with the signal matrix
362     fConvolveColumns(this->signalMatrix, \
363         signalTensorAsArgument);
364
365     // trimming the convolved signal since the signal matrix length remains the same
366     this->signalMatrix = \
367         this->signalMatrix.index({torch::indexing::Slice(0, numsamples), \
368             torch::indexing::Slice()});
369
370     return;
371 }
372
373 /* =====
374 Aim: Decimating basebanded-received signal
375 ----- */
376 void nfdc_decimateSignal(TransmitterClass* transmitterObj){
377
378     // creating the matrix for frequency-shifting
379     torch::Tensor integerArray = torch::linspace(0, \
380         this->signalMatrix.size(0)-1, \
381         this->signalMatrix.size(0)).reshape({this->signalMatrix.size(0),
382             1});
383     integerArray = torch::tile(integerArray, {1, this->num_sensors});
384     integerArray = torch::exp(COMPLEX_1j * transmitterObj->fc * integerArray);
385
386     // storing original number of samples
387     int original_signalMatrix_numsamples = this->signalMatrix.size(0);
388
389     // producing frequency-shifting
390     this->signalMatrix = torch::mul(this->signalMatrix, integerArray);
391
392     // low-pass filter
393     torch::Tensor lowpassfilter_impulseresponse = \
394         this->lowpassFilterCoefficientsForDecimation.reshape(\
395         {this->lowpassFilterCoefficientsForDecimation.numel(), \
396             1});
397     lowpassfilter_impulseresponse = \
398         torch::tile(lowpassfilter_impulseresponse, \
399         {1, this->signalMatrix.size(1)});
400
401     // low-pass filtering the signal
402     fConvolveColumns(this->signalMatrix,
403         lowpassfilter_impulseresponse);
404
405     // Cutting down the extra-samples from convolution
406     this->signalMatrix = \
407         this->signalMatrix.index({torch::indexing::Slice(0, original_signalMatrix_numsamples), \
408             torch::indexing::Slice()});
409
410     // // Cutting off samples in the front.
411     // int cutoffpoint = lowpassfilter_impulseresponse.size(0) - 1;
412     // this->signalMatrix = \
413     //     this->signalMatrix.index({ \
414     //         torch::indexing::Slice(cutoffpoint, \
415     //             torch::indexing::None), \
416     //         torch::indexing::Slice() \
417     //     });
418
419     // building parameters for downsampling
420     int decimation_factor = std::floor(this->sampling_frequency/transmitterObj->bandwidth);
421     this->decimation_factor = decimation_factor;
422     int numsamples_after_decimation = std::floor(this->signalMatrix.size(0)/decimation_factor);

```

```

423 // building the samples which will be subsetting
424 torch::Tensor samplingIndices = \
425     torch::linspace(0, \
426         numsamples_after_decimation * decimation_factor - 1, \
427         numsamples_after_decimation).to(torch::kInt);
428
429 // downsampling the low-pass filtered signal
430 this->signalMatrix = \
431     this->signalMatrix.index({samplingIndices, \
432         torch::indexing::Slice()});
433
434 // returning
435 return;
436 }
437
438 /* =====
439 Aim: Match-filtering
440 ----- */
441 void nfdc_matchFilterDecimatedSignal(){
442
443     // Creating a 2D matrix out of the signal
444     torch::Tensor matchFilter2DMatrix = \
445         this->matchFilter.reshape({this->matchFilter.numel(), 1});
446     matchFilter2DMatrix = \
447         torch::tile(matchFilter2DMatrix, \
448             {1, this->num_sensors});
449
450
451     // 2D convolving to produce the match-filtering
452     fConvolveColumns(this->signalMatrix, \
453         matchFilter2DMatrix);
454
455
456     // Trimming the signal to contain just the signals that make sense to us
457     int startingpoint = matchFilter2DMatrix.size(0) - 1;
458     this->signalMatrix = \
459         this->signalMatrix.index({ \
460             torch::indexing::Slice(startingpoint, \
461                 torch::indexing::None), \
462             torch::indexing::Slice()});
463
464     // // trimming the two ends of the signal
465     // int startingpoint = matchFilter2DMatrix.size(0) - 1;
466     // int endingpoint = this->signalMatrix.size(0) \
467     //     - matchFilter2DMatrix.size(0) \
468     //     + 1;
469     // this->signalMatrix = \
470     //     this->signalMatrix.index({ \
471     //         torch::indexing::Slice(startingpoint, \
472     //             endingpoint), \
473     //         torch::indexing::Slice()});
474
475 }
476
477 /* =====
478 Aim: precompute delay-matrices
479 ----- */
480 void nfdc_precomputeDelayMatrices(TransmitterClass* transmitterObj){
481
482     // calculating range-related parameters
483     int number_of_range_cells = \
484         std::ceil(((this->recording_period * 1500)/2)/this->range_cell_size);
485     int number_of_azimuths_to_image = \
486         std::ceil(transmitterObj->azimuthal_beamwidth / this->azimuth_cell_size);
487
488     // creating centers of range-cell centers
489     torch::Tensor range_centers = \
490         this->range_cell_size * \
491         torch::linspace(0, \
492             number_of_range_cells-1, \
493             number_of_range_cells).to(torch::kFloat) + \
494         this->range_cell_size/2;

```

```

496     this->range_centers = range_centers;
497
498     // creating discretized azimuth-centers
499     torch::Tensor azimuth_centers = \
500         this->azimuth_cell_size * \
501         torch::linspace(0, \
502             number_of_azimuths_to_image - 1, \
503             number_of_azimuths_to_image) + \
504         this->azimuth_cell_size/2;
505     this->azimuth_centers = azimuth_centers;
506
507     // finding the mesh values
508     auto range_azimuth_meshgrid = \
509         torch::meshgrid({range_centers, azimuth_centers}, "ij");
510     torch::Tensor range_grid = range_azimuth_meshgrid[0]; // the columns are range_centers
511     torch::Tensor azimuth_grid = range_azimuth_meshgrid[1]; // the rows are azimuth-centers
512
513     // going from 2D to 3D
514     range_grid = torch::tile(range_grid.reshape({range_grid.size(0), \
515         range_grid.size(1), \
516         1}), \
517         {1,1,this->num_sensors});
518     azimuth_grid = torch::tile(azimuth_grid.reshape({azimuth_grid.size(0), \
519         azimuth_grid.size(1), \
520         1}), \
521         {1, 1, this->num_sensors});
522
523     // creating x_m tensor
524     torch::Tensor sensorCoordinatesSlot = \
525         this->inter_element_spacing * \
526         torch::linspace(0, \
527             this->num_sensors - 1, \
528             this->num_sensors).reshape({1,1,this->num_sensors}).to(torch::kFloat);
529     sensorCoordinatesSlot = \
530         torch::tile(sensorCoordinatesSlot, \
531             {range_grid.size(0), \
532             range_grid.size(1), \
533             1});
534     if(DEBUG_ULA)
535         std::cout << "\t sensorCoordinatesSlot.sizes().vec() = " \
536             << sensorCoordinatesSlot.sizes().vec() \
537             << std::endl;
538
539     // calculating distances
540     torch::Tensor distanceMatrix = \
541         torch::square(range_grid - sensorCoordinatesSlot) + \
542         torch::mul((2 * sensorCoordinatesSlot), \
543             torch::mul(range_grid, \
544                 1 - torch::cos(azimuth_grid * PI/180)));
545     distanceMatrix = torch::sqrt(distanceMatrix);
546
547     // finding the time taken
548     torch::Tensor timeMatrix = distanceMatrix/1500;
549     torch::Tensor sampleMatrix = timeMatrix * this->sampling_frequency;
550
551     // finding the delay to be given
552     auto bruh390 = torch::max(sampleMatrix, 2, true);
553     torch::Tensor max_delay = std::get<0>(bruh390);
554     torch::Tensor delayMatrix = max_delay - sampleMatrix;
555
556     // now that we have the delay entries, we need to create the matrix that does it
557     int decimation_factor = \
558         std::floor(static_cast<float>(this->sampling_frequency)/transmitterObj->bandwidth);
559     this->decimation_factor = decimation_factor;
560
561
562     // calculating frame-size
563     int frame_size = \
564         std::ceil(static_cast<float>((2 * this->range_cell_size / 1500) * \
565             static_cast<float>(this->sampling_frequency)/decimation_factor));
566     this->frame_size = frame_size;
567
568     // // calculating the buffer-zeros to add

```

```

569 // int num_buffer_zeros_per_frame = \
570 //     static_cast<float>>(this->num_sensors - 1) * \
571 //     static_cast<float>>(this->inter_element_spacing) * \
572 //     this->sampling_frequency / 1500;
573
574 int num_buffer_zeros_per_frame = \
575     std::ceil((this->num_sensors - 1) * \
576             this->inter_element_spacing * \
577             this->sampling_frequency \
578             / (1500 * this->decimation_factor));
579
580 // storing to class member
581 this->num_buffer_zeros_per_frame = \
582     num_buffer_zeros_per_frame;
583
584 // calculating the total frame-size
585 int total_frame_size = \
586     this->frame_size + this->num_buffer_zeros_per_frame;
587
588 // creating the multiplication matrix
589 torch::Tensor mulFFTMMatrix = \
590     torch::linspace(0, \
591                    total_frame_size-1, \
592                    total_frame_size).reshape({1, \
593                                                total_frame_size, \
594                                                1}).to(torch::kFloat); // creating an array
595 //                                     1,...,frame_size of shape [1,frame_size, 1];
596
597 mulFFTMMatrix = \
598     torch::div(mulFFTMMatrix, \
599               torch::tensor(total_frame_size).to(torch::kFloat)); // dividing by N
600 mulFFTMMatrix = mulFFTMMatrix * 2 * PI * -1 * COMPLEX_1j; // creating tenosr values for -1j * 2pi * k/N
601 mulFFTMMatrix = \
602     torch::tile(mulFFTMMatrix, \
603                 {number_of_range_cells * number_of_azimuths_to_image, \
604                  1, \
605                  this->num_sensors}); // creating the larger tensor for it
606
607 // populating the matrix
608 for(int azimuth_index = 0; \
609     azimuth_index < number_of_azimuths_to_image; \
610     ++azimuth_index){
611     for(int range_index = 0; \
612         range_index < number_of_range_cells; \
613         ++range_index){
614         // finding the delays for sensors
615         torch::Tensor currentSensorDelays = \
616             delayMatrix.index({range_index, \
617                               azimuth_index, \
618                               torch::indexing::Slice()});
619         // reshaping it to the target size
620         currentSensorDelays = \
621             currentSensorDelays.reshape({1, \
622                                         1, \
623                                         this->num_sensors});
624         // tiling across the plane
625         currentSensorDelays = \
626             torch::tile(currentSensorDelays, \
627                         {1, total_frame_size, 1});
628         // multiplying across the appropriate plane
629         int index_to_place_at = \
630             azimuth_index * number_of_range_cells + \
631             range_index;
632         mulFFTMMatrix.index_put_({index_to_place_at, \
633                                   torch::indexing::Slice(), \
634                                   torch::indexing::Slice()}, \
635                                 currentSensorDelays);
636     }
637 }
638
639 // storing the mulFFTMMatrix
640 this->mulFFTMMatrix = mulFFTMMatrix;

```



```

640 }
641
642 /* =====
643 Aim: Beamforming the signal
644 ===== */
645 void nfdc_beamforming(TransmitterClass* transmitterObj){
646
647     // ensuring the signal matrix is in the shape we want
648     if(this->signalMatrix.size(1) != this->num_sensors)
649         throw std::runtime_error("The second dimension doesn't correspond to the number of sensors \n");
650
651     // adding the batch-dimension
652     this->signalMatrix = \
653         this->signalMatrix.reshape({          \
654             1,                                \
655             this->signalMatrix.size(0),        \
656             this->signalMatrix.size(1)});
657
658     // zero-padding to ensure correctness
659     int ideal_length = \
660         std::ceil(this->range_centers.numel() * this->frame_size);
661     int num_zeros_to_pad_signal_along_dimension_0 = \
662         ideal_length - this->signalMatrix.size(1);
663
664     // printing
665     if (DEBUG_ULA) PRINTSMALLLINE
666     if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->range_centers.numel()      =
667         "<<this->range_centers.numel() <<std::endl;
668     if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->frame_size                =
669         "<<this->frame_size <<std::endl;
670     if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | ideal_length                =
671         "<<ideal_length <<std::endl;
672     if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->signalMatrix.size(1)      =
673         "<<this->signalMatrix.size(1) <<std::endl;
674     if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | num_zeros_to_pad_signal_along_dimension_0
675         = "<<num_zeros_to_pad_signal_along_dimension_0 <<std::endl;
676     if (DEBUG_ULA) PRINTSPACE
677
678     // appending or slicing based on the requirements
679     if (num_zeros_to_pad_signal_along_dimension_0 <= 0) {
680
681         // sending out a warning that slicing is going on
682         if (DEBUG_ULA) std::cerr <<"\t\t ULAClass::nfdc_beamforming | Please note that the signal matrix
683             has been sliced. This could lead to loss of information"<<std::endl;
684
685         // slicing the signal matrix
686         if (DEBUG_ULA) PRINTSPACE
687         if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->signalMatrix.shape (before
688             slicing) = "<< this->signalMatrix.sizes().vec() <<std::endl;
689         this->signalMatrix = \
690             this->signalMatrix.index({torch::indexing::Slice(), \
691                 torch::indexing::Slice(0, ideal_length), \
692                 torch::indexing::Slice()});
693         if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->signalMatrix.shape (after
694             slicing) = "<< this->signalMatrix.sizes().vec() <<std::endl;
695         if (DEBUG_ULA) PRINTSPACE
696     }
697     else {
698         // creating a zero-filled tensor to append to signal matrix
699         torch::Tensor zero_tensor = \
700             torch::zeros({this->signalMatrix.size(0), \
701                 num_zeros_to_pad_signal_along_dimension_0, \
702                 this->num_sensors}).to(torch::kFloat);
703
704         // appending to signal matrix
705         this->signalMatrix = \
706             torch::cat({this->signalMatrix, zero_tensor}, 1);
707     }
708
709     // breaking the signal into frames
710     fBuffer2D(this->signalMatrix, frame_size);

```



```

704
705 // add some zeros to the end of frames to accomodate delaying of signals.
706 torch::Tensor zero_filled_tensor = \
707     torch::zeros({this->signalMatrix.size(0), \
708                 this->num_buffer_zeros_per_frame, \
709                 this->num_sensors}).to(torch::kFloat);
710 this->signalMatrix = \
711     torch::cat({this->signalMatrix, \
712                zero_filled_tensor}, 1);
713
714 // tiling it to ensure that it works for all range-angle combinations
715 int number_of_azimuths_to_image = this->azimuth_centers.numel();
716 this->signalMatrix = \
717     torch::tile(this->signalMatrix, \
718                 {number_of_azimuths_to_image, 1, 1});
719
720 // element-wise multiplying the signals to delay each of the frame accordingly
721 this->signalMatrix = torch::mul(this->signalMatrix, \
722                                this->mulFFTMatrix);
723
724 // summing up the signals
725 // this->signalMatrix = torch::sum(this->signalMatrix, \
726 //                                2, \
727 //                                true);
728 this->signalMatrix = torch::sum(this->signalMatrix, \
729                                2, \
730                                false);
731
732 // printing some stuff
733 if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming: this->azimuth_centers.numel() =
734     "<<this->azimuth_centers.numel() <<std::endl;
735 if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming: this->range_centers.numel() =
736     "<<this->range_centers.numel() <<std::endl;
737 if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming: total number =
738     "<<this->range_centers.numel() * this->azimuth_centers.numel() <<std::endl;
739 if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming: this->signalMatrix.sizes().vec() =
740     "<<this->signalMatrix.sizes().vec() <<std::endl;
741
742 // creating a tensor to store the final image
743 torch::Tensor finalImage = \
744     torch::zeros({this->frame_size * this->range_centers.numel(), \
745                  this->azimuth_centers.numel()}).to(torch::kComplexFloat);
746
747 // creating a loop to assign values
748 for(int range_index = 0; range_index < this->range_centers.numel(); ++range_index){
749     for(int angle_index = 0; angle_index < this->azimuth_centers.numel(); ++angle_index){
750
751         // getting row index
752         int rowindex = \
753             angle_index * this->range_centers.numel() \
754             + range_index;
755
756         // getting the strip to store
757         torch::Tensor strip = \
758             this->signalMatrix.index({rowindex, \
759                                     torch::indexing::Slice()});
760
761         // taking just the first few values
762         strip = strip.index({torch::indexing::Slice(0, this->frame_size)});
763
764         // placing the strips on the image
765         finalImage.index_put_({\
766             torch::indexing::Slice((range_index)*this->frame_size, \
767                                     (range_index+1)*this->frame_size), \
768             angle_index}, \
769             strip);
770     }
771 }
772
773 // saving the image
774 this->beamformedImage = finalImage;

```

```

773
774 // converting image from polar to cartesian
775 nfdc_PolarToCartesian();
776 std::cout<<"\t\t ULAClass::nfdc_beamforming: finished nfdc_PolarToCartesian"<<std::endl;
777
778 }
779
780 /* =====
781 Aim: Converting Polar Image to Cartesian
782 .....
783 Note:
784 > For now, we're assuming that the r value is one.
785 ----- */
786 void nfdc_PolarToCartesian(){
787
788 // deciding image dimensions
789 int num_pixels_width = 512;
790 int num_pixels_height = 512;
791
792 // creating query points
793 torch::Tensor max_right = \
794     torch::cos( \
795         torch::max( \
796             this->azimuth_centers \
797             - torch::mean(this->azimuth_centers) \
798             + torch::tensor({90}).to(torch::kFloat)) \
799         * PI/180);
800 torch::Tensor max_left = \
801     torch::cos( \
802         torch::min(this->azimuth_centers \
803             - torch::mean(this->azimuth_centers) \
804             + torch::tensor({90}).to(torch::kFloat)) \
805         * PI/180);
806 torch::Tensor max_top = torch::tensor({1});
807 torch::Tensor max_bottom = torch::min(this->range_centers);
808
809
810
811 // creating query points along the x-dimension
812 torch::Tensor query_x = \
813     torch::linspace( \
814         max_left, \
815         max_right, \
816         num_pixels_width \
817         ).to(torch::kFloat);
818
819 torch::Tensor query_y = \
820     torch::linspace( \
821         max_bottom.item<float>(), \
822         max_top.item<float>(), \
823         num_pixels_height \
824         ).to(torch::kFloat);
825
826
827 // converting original coordinates to their corresponding cartesian
828 float delta_r = 1/static_cast<float>(this->beamformedImage.size(0));
829 float delta_azimuth = \
830     torch::abs( \
831         this->azimuth_centers.index({1}) \
832         - this->azimuth_centers.index({0}) \
833         ).item<float>();
834
835
836
837 // getting query points
838 torch::Tensor range_values = \
839     torch::linspace( \
840         delta_r, \
841         this->beamformedImage.size(0) * delta_r, \
842         this->beamformedImage.size(0) \
843         ).to(torch::kFloat);
844 range_values = \
845     range_values.reshape({range_values.numel(), 1});

```

```

846 range_values = \
847     torch::tile(range_values, \
848         {1, this->azimuth_centers.numel()});
849
850 // getting angle-values
851 torch::Tensor angle_values = \
852     this->azimuth_centers \
853     - torch::mean(this->azimuth_centers) \
854     + torch::tensor({90});
855 angle_values = \
856     torch::tile( \
857         angle_values, \
858         {this->beamformedImage.size(0), 1});
859
860
861 // converting to cartesian original points
862 torch::Tensor query_original_x = \
863     range_values * torch::cos(angle_values * PI/180);
864 torch::Tensor query_original_y = \
865     range_values * torch::sin(angle_values * PI/180);
866
867 // converting points to vector 2D format
868 torch::Tensor query_source = \
869     torch::cat({ \
870         query_original_x.reshape({1, query_original_x.numel()}), \
871         query_original_y.reshape({1, query_original_y.numel()}), \
872         0);
873
874 // converting reflectivity to corresponding 2D format
875 torch::Tensor reflectivity_vectors = \
876     this->beamformedImage.reshape({1, this->beamformedImage.numel()});
877
878 // creating image
879 int num_pixels_x = 512;
880 int num_pixels_y = 512;
881 torch::Tensor cartesianImageLocal = \
882     torch::zeros({num_pixels_x, num_pixels_y}).to(torch::kComplexFloat);
883
884
885 /*
886 Next Aim: start interpolating the points on the uniform grid.
887 */
888
889 for(int x_index = 0; x_index < query_x.numel(); ++x_index){
890     std::cout<<"\t\t\t x_index = "<<x_index<<std::endl;
891     for(int y_index = 0; y_index < query_y.numel(); ++y_index){
892
893         // getting current values
894         torch::Tensor current_x = query_x.index({x_index}).reshape({1, 1});
895         torch::Tensor current_y = query_y.index({y_index}).reshape({1, 1});
896
897         // getting the query value
898         torch::Tensor query_vector = torch::cat({current_x, current_y}, 0);
899
900         // copying the query source
901         torch::Tensor query_source_relative = query_source;
902         query_source_relative = query_source_relative - query_vector;
903
904         // subsetting using absolute values and masks
905         float threshold = delta_r * 10;
906         // PRINTDOTS
907         auto mask_row = \
908             torch::abs(query_source_relative[0]) <= threshold;
909         auto mask_col = \
910             torch::abs(query_source_relative[1]) <= threshold;
911         auto mask_together = torch::mul(mask_row, mask_col);
912
913         // calculating number of points in threshold neighbourhood
914         int num_points_in_threshold_neighbourhood = \
915             torch::sum(mask_together).item<int>();
916         if (num_points_in_threshold_neighbourhood == 0){
917             continue;
918         }

```

```

919     }
920
921     // subsetting points in neighbourhood
922     torch::Tensor PointsInNeighbourhood = \
923         query_source_relative.index({
924             torch::indexing::Slice(), \
925             mask_together});
926     torch::Tensor ReflectivitiesInNeighbourhood = \
927         reflectivity_vectors.index({torch::indexing::Slice(), mask_together});
928
929     // finding the distance between the points
930     torch::Tensor relativeDistances = \
931         torch::linalg_norm(PointsInNeighbourhood, 2, 0, true, torch::kFloat);
932
933     // calculating weighing factor
934     torch::Tensor weighingFactor = \
935         torch::nn::functional::softmax( \
936             torch::max(relativeDistances)- relativeDistances, \
937             torch::nn::functional::SoftmaxFuncOptions(1));
938
939     // combining intensities using distances
940     torch::Tensor finalIntensity = \
941         torch::sum(
942             torch::mul(weighingFactor, \
943                 ReflectivitiesInNeighbourhood));
944
945     // assigning values
946     cartesianImageLocal.index_put_({x_index, y_index}, finalIntensity);
947
948 }
949 }
950
951 // saving to member function
952 this->cartesianImage = cartesianImageLocal;
953
954 }
955
956 /* =====
957 Aim: create acoustic image directly
958 ===== */
959 void nfdc_createAcousticImage(ScattererClass* scatterers, \
960     TransmitterClass* transmitterObj){
961
962     // first we ensure that the scatterers are in our frame of reference
963     scatterers->coordinates = scatterers->coordinates - this->location;
964
965     // finding the spherical coordinates of the scatterers
966     torch::Tensor scatterers_spherical = fCart2Sph(scatterers->coordinates);
967
968     // note that its not precisely projection. its rotation. So the original lengths must be maintained.
969     // but thats easy since the operation of putting the elevation to be zero works just fine.
970     scatterers_spherical.index_put_({1, torch::indexing::Slice()}, 0);
971
972     // converting the points back to cartesian
973     torch::Tensor scatterers_acoustic_cartesian = fSph2Cart(scatterers_spherical);
974
975     // removing the z-dimension
976     scatterers_acoustic_cartesian = \
977         scatterers_acoustic_cartesian.index({torch::indexing::Slice(0, 2, 1), \
978             torch::indexing::Slice()});
979
980     // deciding image dimensions
981     int num_pixels_x = 512;
982     int num_pixels_y = 512;
983     torch::Tensor acousticImage = \
984         torch::zeros({num_pixels_x, \
985             num_pixels_y}).to(torch::kFloat);
986
987     // finding the max and min values
988     torch::Tensor min_x = torch::min(scatterers_acoustic_cartesian[0]);
989     torch::Tensor max_x = torch::max(scatterers_acoustic_cartesian[0]);
990     torch::Tensor min_y = torch::min(scatterers_acoustic_cartesian[1]);
991     torch::Tensor max_y = torch::max(scatterers_acoustic_cartesian[1]);

```

```

991
992 // creating query grids
993 torch::Tensor query_x = torch::linspace(0, 1, num_pixels_x);
994 torch::Tensor query_y = torch::linspace(0, 1, num_pixels_y);
995
996 // scaling it up to image max-point spread
997 query_x = min_x + (max_x - min_x) * query_x;
998 query_y = min_y + (max_y - min_y) * query_y;
999 float delta_queryx = (query_x[1] - query_x[0]).item<float>();
1000 float delta_queryy = (query_y[1] - query_y[0]).item<float>();
1001
1002 // creating a mesh-grid
1003 auto queryMeshGrid = torch::meshgrid({query_x, query_y}, "ij");
1004 query_x = queryMeshGrid[0].reshape({1, queryMeshGrid[0].numel()});
1005 query_y = queryMeshGrid[1].reshape({1, queryMeshGrid[1].numel()});
1006 torch::Tensor queryMatrix = torch::cat({query_x, query_y}, 0);
1007
1008 // printing shapes
1009 if(DEBUG_ULA) PRINTSMALLLINE
1010 if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: query_x.shape =
    "<<query_x.sizes().vec()<<std::endl;
1011 if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: query_y.shape =
    "<<query_y.sizes().vec()<<std::endl;
1012 if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: queryMatrix.shape =
    "<<queryMatrix.sizes().vec()<<std::endl;
1013
1014 // setting up threshold values
1015 float threshold_value = \
1016     std::min(delta_queryx, \
1017         delta_queryy); if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line
    711"<<std::endl;
1018
1019 // putting a loop through the whole thing
1020 for(int i = 0; i<queryMatrix[0].numel(); ++i){
1021     // for each element in the query matrix
1022     if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 716"<<std::endl;
1023
1024     // calculating relative position of all the points
1025     torch::Tensor relativeCoordinates = \
1026         scatterers_acoustic_cartesian - \
1027         queryMatrix.index({torch::indexing::Slice(), i}).reshape({2, 1}); if(DEBUG_ULA)
        std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 720"<<std::endl;
1028
1029     // calculating distances between all the points and the query point
1030     torch::Tensor relativeDistances = \
1031         torch::linalg_norm(relativeCoordinates, \
1032             1, 0, true, \
1033             torch::kFloat);if(DEBUG_ULA) std::cout<<"\t\t\t
        ULAClass::nfdc_createAcousticImage: line 727"<<std::endl;
1034
1035     // finding points that are within the threshold
1036     torch::Tensor conditionMeetingPoints = \
        relativeDistances.squeeze() <= threshold_value;if(DEBUG_ULA) std::cout<<"\t\t\t
        ULAClass::nfdc_createAcousticImage: line 729"<<std::endl;
1037
1038     // subsetting the points in the neighbourhood
1039     if(torch::sum(conditionMeetingPoints).item<float>() == 0){
1040
1041         // continuing implementation if there are no points in the neighbourhood
1042         continue; if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line
        735"<<std::endl;
1043     }
1044     else{
1045         // creating mask for points in the neighbourhood
1046         auto mask = (conditionMeetingPoints == 1);if(DEBUG_ULA) std::cout<<"\t\t\t
        ULAClass::nfdc_createAcousticImage: line 739"<<std::endl;
1047
1048         // subsetting relative distances in the neighbourhood
1049         torch::Tensor distanceInTheNeighbourhood = \
1050             relativeDistances.index({torch::indexing::Slice(), mask});if(DEBUG_ULA) std::cout<<"\t\t\t
        ULAClass::nfdc_createAcousticImage: line 743"<<std::endl;
1051
1052         // subsetting reflectivity of points in the neighbourhood
1053         torch::Tensor reflectivityInTheNeighbourhood = \

```

```

1054         scatterers->reflectivity.index({torch::indexing::Slice(), mask});if(DEBUG_ULA)
1055             std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 747"<<std::endl;
1056
1057         // assigning intensity as a function of distance and reflectivity
1058         torch::Tensor reflectivityAssignment = \
1059             torch::mul(torch::exp(-distanceInTheNeighbourhood), \
1060                 reflectivityInTheNeighbourhood);if(DEBUG_ULA) std::cout<<"\t\t\t
1061                 ULAClass::nfdc_createAcousticImage: line 752"<<std::endl;
1062         reflectivityAssignment = \
1063             torch::sum(reflectivityAssignment);if(DEBUG_ULA) std::cout<<"\t\t\t
1064                 ULAClass::nfdc_createAcousticImage: line 754"<<std::endl;
1065
1066         // assigning this value to the image pixel intensity
1067         int pixel_position_x = i*num_pixels_x;
1068         int pixel_position_y = std::floor(i/num_pixels_x);
1069         acousticImage.index_put_({pixel_position_x, \
1070             pixel_position_y}, \
1071             reflectivityAssignment.item<float>());if(DEBUG_ULA) std::cout<<"\t\t\t
1072             ULAClass::nfdc_createAcousticImage: line 761"<<std::endl;
1073     }
1074 }
1075
1076 // storing the acoustic-image to the member
1077 this->currentArtificialAcousticImage = acousticImage;
1078
1079 // // saving the torch::tensor
1080 // torch::save(acousticImage, \
1081 //     "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/Assets/acoustic_image.pt");
1082
1083 // // bringing it back to the original coordinates
1084 // scatterers->coordinates = scatterers->coordinates + this->location;
1085 }
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122

```

```
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132 }
```

---

#### 8.1.4 Class: Autonomous Underwater Vehicle

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

```

1 // including header-files
2 #include "ScattererClass.h"
3 #include "TransmitterClass.h"
4 #include "ULAClass.h"
5 #include <iostream>
6 #include <ostream>
7 #include <torch/torch.h>
8 #include <cmath>
9
10 // including functions
11 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fGetCurrentTimeFormatted.cpp"
12 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fCart2Sph.cpp"
13
14 #pragma once
15
16 // function to plot the thing
17 void fPlotTensors(){
18     system("python /Users/vrsreeganesh/Documents/GitHub/AUV/Code/Python/TestingSaved_tensors.py");
19 }
20
21
22 void fSaveSeafloorScatteres(ScattererClass scatterer, \
23                             ScattererClass scatterer_fls, \
24                             ScattererClass scatterer_port, \
25                             ScattererClass scatterer_starboard){
26
27     // saving the ground-truth
28     ScattererClass SeafloorScatter_gt = scatterer;
29     torch::save(SeafloorScatter_gt.coordinates,
30                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt.pt");
31     torch::save(SeafloorScatter_gt.reflectivity,
32                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt_reflectivity.pt");
33
34     // saving coordinates
35     torch::save(scatterer_fls.coordinates,
36                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_fls_coordinates.pt");
37     torch::save(scatterer_port.coordinates,
38                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_port_coordinates.pt");
39     torch::save(scatterer_starboard.coordinates,
40                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_starboard.coordinates.pt");
41
42     // saving reflectivities
43     torch::save(scatterer_fls.reflectivity,
44                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_fls_coordinates_reflectivity.pt");
45     torch::save(scatterer_port.reflectivity,
46                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_port_coordinates_reflectivity.pt");
47     torch::save(scatterer_starboard.reflectivity,
48                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_starboard.coordinates_reflectivity.pt");
49
50     // plotting tensors
51     fPlotTensors();
52
53     // // saving the tensors
54     // if (true) {
55
56     //     // getting time ID
57     //     auto timeID = fGetCurrentTimeFormatted();
58
59     //     std::cout<<"\t\t\t\t\t Saving Tensors (timeID: "<<timeID<<)"<<std::endl;
60
61     //     // saving the ground-truth
62     //     ScattererClass SeafloorScatter_gt = scatterer;
63     //     torch::save(SeafloorScatter_gt.coordinates, \
64     //                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt.pt");
65     //     torch::save(SeafloorScatter_gt.reflectivity, \
66     //                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt_reflectivity.pt");
67
68     // }
69 }

```





```

126 TransmitterClass transmitter_starboard; // mounted transmitter [obj of class, TransmitterClass]
127
128 // derived or dependent attributes
129 torch::Tensor signalMatrix_1; // matrix containing the signals obtained from ULA_1
130 torch::Tensor largeSignalMatrix_1; // matrix holding signal of synthetic aperture
131 torch::Tensor beamformedLargeSignalMatrix; // each column is the beamformed signal at each stop-hop
132
133 // plotting mode
134 bool plottingmode; // to suppress plotting associated with classes
135
136 // spotlight mode related
137 torch::Tensor absolute_coords_patch_cart; // cartesian coordinates of patch
138
139 // Synthetic Aperture Related
140 torch::Tensor ApertureSensorLocations; // sensor locations of aperture
141
142
143
144
145
146
147
148
149 /* =====
150 Aim: Init
151 -----*/
152 void init(){
153
154     // call sync-component attributes
155     this->syncComponentAttributes();
156
157     // initializing all the ULAs
158     this->ULA_fls.init( &this->transmitter_fls);
159     this->ULA_port.init( &this->transmitter_port);
160     this->ULA_starboard.init( &this->transmitter_starboard);
161
162     // precomputing delay-matrices for the ULA-class
163     std::thread ULA_fls_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
164                                             &this->ULA_fls, \
165                                             &this->transmitter_fls);
166     std::thread ULA_port_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
167                                             &this->ULA_port, \
168                                             &this->transmitter_port);
169     std::thread ULA_starboard_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
170                                                  &this->ULA_starboard, \
171                                                  &this->transmitter_starboard);
172
173     // joining the threads back
174     ULA_fls_precompute_weights_t.join();
175     ULA_port_precompute_weights_t.join();
176     ULA_starboard_precompute_weights_t.join();
177 }
178
179
180
181
182 /*=====
183 Aim: stepping motion
184 -----*/
185 void step(float timestep){
186
187     // updating location
188     this->location = this->location + this->velocity * timestep;
189     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 81 \n";
190
191     // updating attributes of members
192     this->syncComponentAttributes();
193     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 85 \n";
194 }
195
196
197
198 /*=====

```

```

199 Aim: updateAttributes
200 -----*/
201 void syncComponentAttributes(){
202
203     // updating ULA attributes
204     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 97 \n";
205
206     // updating locations
207     this->ULA_fls.location      = this->location;
208     this->ULA_port.location     = this->location;
209     this->ULA_starboard.location = this->location;
210
211     // updating the pointing direction of the ULAs
212     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 99 \n";
213     torch::Tensor ula_fls_sensor_direction_spherical = fCart2Sph(this->pointing_direction);    //
214     spherical coords
215     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 101 \n";
216     ula_fls_sensor_direction_spherical[0]           = ula_fls_sensor_direction_spherical[0] - 90;
217     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 98 \n";
218     torch::Tensor ula_fls_sensor_direction_cart     = fSph2Cart(ula_fls_sensor_direction_spherical);
219     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 100 \n";
220
221     this->ULA_fls.sensorDirection      = ula_fls_sensor_direction_cart; // assigning sensor direction for
222     ULA-FLS
223     this->ULA_port.sensorDirection     = -this->pointing_direction;    // assigning sensor direction for
224     ULA-Port
225     this->ULA_starboard.sensorDirection = -this->pointing_direction;    // assigning sensor direction for
226     ULA-Starboard
227     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 105 \n";
228
229     // // calling the function to update the arguments
230     // this->ULA_fls.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line
231     109 \n";
232     // this->ULA_port.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line
233     111 \n";
234     // this->ULA_starboard.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) std::cout<<"\t AUVClass:
235     line 113 \n";
236
237     // updating transmitter locations
238     this->transmitter_fls.location = this->location;
239     this->transmitter_port.location = this->location;
240     this->transmitter_starboard.location = this->location;
241     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 102 \n";
242
243     // updating transmitter pointing directions
244     this->transmitter_fls.updatePointingAngle( this->pointing_direction);
245     this->transmitter_port.updatePointingAngle( this->pointing_direction);
246     this->transmitter_starboard.updatePointingAngle( this->pointing_direction);
247     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 108 \n";
248 }
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
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
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

265
266 // calling the method associated with the transmitter
267 if(DEBUGMODE_AUV) {std::cout<<"\t\t scatterers.shape = "; fPrintTensorSize(scatterers->coordinates);}
268 if(DEBUGMODE_AUV) std::cout<<"\t\t tilt_angle = "<<tilt_angle<<std::endl;
269 transmitterObj->subsetScatterers(scatterers, tilt_angle);
270 if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 124 \n";
271 }
272
273 // yaw-correction matrix
274 torch::Tensor createYawCorrectionMatrix(torch::Tensor pointing_direction_spherical, \
275                                         float target_azimuth_deg){
276
277 // building parameters
278 torch::Tensor azimuth_correction =
279     torch::tensor({target_azimuth_deg}).to(torch::kFloat).to(DEVICE) - \
280     pointing_direction_spherical[0];
281 torch::Tensor azimuth_correction_radians = azimuth_correction * PI / 180;
282
283 torch::Tensor yawCorrectionMatrix = \
284     torch::tensor({torch::cos(azimuth_correction_radians).item<float>(), \
285                   torch::cos(torch::tensor({90}).to(torch::kFloat).to(DEVICE)*PI/180 +
286                           azimuth_correction_radians).item<float>(), \
287                   (float)0, \
288                   torch::sin(azimuth_correction_radians).item<float>(), \
289                   torch::sin(torch::tensor({90}).to(torch::kFloat).to(DEVICE)*PI/180 +
290                           azimuth_correction_radians).item<float>(), \
291                   (float)0, \
292                   (float)0, \
293                   (float)0, \
294                   (float)1}).reshape({3,3}).to(torch::kFloat).to(DEVICE);
295
296 // returning the matrix
297 return yawCorrectionMatrix;
298 }
299
300 // pitch-correction matrix
301 torch::Tensor createPitchCorrectionMatrix(torch::Tensor pointing_direction_spherical, \
302                                         float target_elevation_deg){
303
304 // building parameters
305 torch::Tensor elevation_correction =
306     torch::tensor({target_elevation_deg}).to(torch::kFloat).to(DEVICE) - \
307     pointing_direction_spherical[1];
308 torch::Tensor elevation_correction_radians = elevation_correction * PI / 180;
309
310 // creating the matrix
311 torch::Tensor pitchCorrectionMatrix = \
312     torch::tensor({(float)1, \
313                   (float)0, \
314                   (float)0, \
315                   (float)0, \
316                   torch::cos(elevation_correction_radians).item<float>(), \
317                   torch::cos(torch::tensor({90}).to(torch::kFloat).to(DEVICE)*PI/180 +
318                           elevation_correction_radians).item<float>(), \
319                   (float)0, \
320                   torch::sin(elevation_correction_radians).item<float>(), \
321                   torch::sin(torch::tensor({90}).to(torch::kFloat).to(DEVICE)*PI/180 +
322                           elevation_correction_radians).item<float>())}.reshape({3,3}).to(torch::kFloat);
323
324 // returning the matrix
325 return pitchCorrectionMatrix;
326 }
327
328 // Signal Simulation
329 void simulateSignal(ScattererClass& scatterer){
330
331 // printing status
332 std::cout << "\t AUVClass::simulateSignal: Began Signal Simulation" << std::endl;
333
334 // making three copies
335 ScattererClass scatterer_fls = scatterer;
336 ScattererClass scatterer_port = scatterer;
337 ScattererClass scatterer_starboard = scatterer;

```

```

332
333
334 // finding the pointing direction in spherical
335 torch::Tensor auv_pointing_direction_spherical = fCart2Sph(this->pointing_direction);
336
337
338 // asking the transmitters to subset the scatterers by multithreading
339 std::thread transmitterFLSSubset_t(&AUVClass::subsetScatterers, this, \
340                                     &scatterer_fls, \
341                                     &this->transmitter_fls, \
342                                     (float)0);
343 std::thread transmitterPortSubset_t(&AUVClass::subsetScatterers, this, \
344                                     &scatterer_port, \
345                                     &this->transmitter_port, \
346                                     auv_pointing_direction_spherical[1].item<float>());
347 std::thread transmitterStarboardSubset_t(&AUVClass::subsetScatterers, this, \
348                                     &scatterer_starboard, \
349                                     &this->transmitter_starboard, \
350                                     - auv_pointing_direction_spherical[1].item<float>());
351
352 // joining the subset threads back
353 transmitterFLSSubset_t.join();
354 transmitterPortSubset_t.join();
355 transmitterStarboardSubset_t.join();
356
357
358 // multithreading the saving tensors part.
359 std::thread savetensor_t(fSaveSeafloorScatterers, \
360                         scatterer, \
361                         scatterer_fls, \
362                         scatterer_port, \
363                         scatterer_starboard);
364
365
366 // asking ULAs to simulate signal through multithreading
367 std::thread ulafls_signalsim_t(&ULAClass::nfdc_simulateSignal, \
368                               &this->ULA_fls, \
369                               &scatterer_fls, \
370                               &this->transmitter_fls);
371 std::thread ulaport_signalsim_t(&ULAClass::nfdc_simulateSignal, \
372                               &this->ULA_port, \
373                               &scatterer_port, \
374                               &this->transmitter_port);
375 std::thread ulastarboard_signalsim_t(&ULAClass::nfdc_simulateSignal, \
376                                     &this->ULA_starboard, \
377                                     &scatterer_starboard, \
378                                     &this->transmitter_starboard);
379
380 // joining them back
381 ulafls_signalsim_t.join(); // joining back the thread for ULA-FLS
382 ulaport_signalsim_t.join(); // joining back the signals-sim thread for ULA-Port
383 ulastarboard_signalsim_t.join(); // joining back the signal-sim thread for ULA-Starboard
384 savetensor_t.join(); // joining back the signal-sim thread for tensor-saving
385
386
387 }
388
389 // Imaging Function
390 void image(){
391
392 // asking ULAs to decimate the signals obtained at each time step
393 std::thread ULA_fls_image_t(&ULAClass::nfdc_decimateSignal, \
394                             &this->ULA_fls, \
395                             &this->transmitter_fls);
396 std::thread ULA_port_image_t(&ULAClass::nfdc_decimateSignal, \
397                             &this->ULA_port, \
398                             &this->transmitter_port);
399 std::thread ULA_starboard_image_t(&ULAClass::nfdc_decimateSignal, \
400                                  &this->ULA_starboard, \
401                                  &this->transmitter_starboard);
402
403 // joining the threads back
404 ULA_fls_image_t.join();

```

```

405     ULA_port_image_t.join();
406     ULA_starboard_image_t.join();
407
408     // saving the decimated signal
409     if (SAVE_DECIMATED_SIGNAL_MATRIX) {
410         std::cout << "\t AUVClass::image: saving decimated signal matrix" \
411             << std::endl;
412         torch::save(this->ULA_fls.signalMatrix, \
413             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/decimated_signalMatrix_fls.pt");
414         torch::save(this->ULA_port.signalMatrix, \
415             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/decimated_signalMatrix_port.pt");
416         torch::save(this->ULA_starboard.signalMatrix, \
417             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/decimated_signalMatrix_starboard.pt");
418     }
419
420     // asking ULAs to match-filter the signals
421     std::thread ULA_fls_matchfilter_t( \
422         &ULAClass::nfdc_matchFilterDecimatedSignal, \
423         &this->ULA_fls);
424     std::thread ULA_port_matchfilter_t( \
425         &ULAClass::nfdc_matchFilterDecimatedSignal, \
426         &this->ULA_port);
427     std::thread ULA_starboard_matchfilter_t( \
428         &ULAClass::nfdc_matchFilterDecimatedSignal, \
429         &this->ULA_starboard);
430
431     // joining the threads back
432     ULA_fls_matchfilter_t.join();
433     ULA_port_matchfilter_t.join();
434     ULA_starboard_matchfilter_t.join();
435
436
437     // saving the decimated signal
438     if (SAVE_MATCHFILTERED_SIGNAL_MATRIX) {
439
440         // saving the tensors
441         std::cout << "\t AUVClass::image: saving match-filtered signal matrix" \
442             << std::endl;
443         torch::save(this->ULA_fls.signalMatrix, \
444             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/matchfiltered_signalMatrix_fls.pt");
445         torch::save(this->ULA_port.signalMatrix, \
446             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/matchfiltered_signalMatrix_port.pt");
447         torch::save(this->ULA_starboard.signalMatrix, \
448             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/matchfiltered_signalMatrix_starboard.pt");
449
450         // running python-script
451     }
452
453
454
455
456     // performing the beamforming
457     std::thread ULA_fls_beamforming_t(&ULAClass::nfdc_beamforming, \
458         &this->ULA_fls, \
459         &this->transmitter_fls);
460     // std::thread ULA_port_beamforming_t(&ULAClass::nfdc_beamforming, \
461     //     &this->ULA_port, \
462     //     &this->transmitter_port);
463     // std::thread ULA_starboard_beamforming_t(&ULAClass::nfdc_beamforming, \
464     //     &this->ULA_starboard, \
465     //     &this->transmitter_starboard);
466
467     // joining the filters back
468     ULA_fls_beamforming_t.join();
469     // ULA_port_beamforming_t.join();
470     // ULA_starboard_beamforming_t.join();
471
472 }
473
474
475
476
477 /* =====

```

```

478 Aim: directly create acoustic image
479 ----- */
480 void createAcousticImage(ScattererClass* scatterers){
481
482     // making three copies
483     ScattererClass scatterer_fls    = scatterers;
484     ScattererClass scatterer_port   = scatterers;
485     ScattererClass scatterer_starboard = scatterers;
486
487     // printing size of scatterers before subsetting
488     PRINTSMALLLINE
489     std::cout<< "\t > AUVClass::createAcousticImage: Beginning Scatterer Subsetting"<<std::endl;
490     std::cout<<"\t AUVClass::createAcousticImage: scatterer_fls.coordinates.shape (before) = ";
491         fPrintTensorSize(scatterer_fls.coordinates);
492     std::cout<<"\t AUVClass::createAcousticImage: scatterer_port.coordinates.shape (before) = ";
493         fPrintTensorSize(scatterer_port.coordinates);
494     std::cout<<"\t AUVClass::createAcousticImage: scatterer_starboard.coordinates.shape (before) = ";
495         fPrintTensorSize(scatterer_starboard.coordinates);
496
497     // finding the pointing direction in spherical
498     torch::Tensor auv_pointing_direction_spherical = fCart2Sph(this->pointing_direction);
499
500     // asking the transmitters to subset the scatterers by multithreading
501     std::thread transmitterFLSSubset_t(&AUVClass::subsetScatterers, this, \
502         &scatterer_fls,\
503         &this->transmitter_fls, \
504         (float)0);
505     std::thread transmitterPortSubset_t(&AUVClass::subsetScatterers, this, \
506         &scatterer_port,\
507         &this->transmitter_port, \
508         auv_pointing_direction_spherical[1].item<float>());
509     std::thread transmitterStarboardSubset_t(&AUVClass::subsetScatterers, this, \
510         &scatterer_starboard, \
511         &this->transmitter_starboard, \
512         - auv_pointing_direction_spherical[1].item<float>());
513
514     // joining the subset threads back
515     transmitterFLSSubset_t.join( );
516     transmitterPortSubset_t.join( );
517     transmitterStarboardSubset_t.join( );
518
519     // asking the ULAs to directly create acoustic images
520     std::thread ULA_fls_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, this->ULA_fls, \
521         &scatterer_fls, &this->transmitter_fls);
522     std::thread ULA_port_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, &this->ULA_port, \
523         &scatterer_port, &this->transmitter_port);
524     std::thread ULA_starboard_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, &this->ULA_starboard, \
525         &scatterer_starboard, &this->transmitter_starboard);
526
527     // joining the threads back
528     ULA_fls_acoustic_image_t.join( );
529     ULA_port_acoustic_image_t.join( );
530     ULA_starboard_acoustic_image_t.join();
531 }
532
533 };
534
535
536
537
538
539
540
541
542
543
544
545
546
547

```

```
548
549
550
551
552
553
554
555
556 // 0.0000,
557 // 0.0000,
558 // 0.0000,
559 // 0.0000,
560 // 0.0000,
561 // 0.0000,
562 // 0.0000,
563 // 0.0000,
564 // 0.0000,
565 // 0.0000,
566 // 0.0000,
567 // 0.0000,
568 // 0.0000,
569 // 0.0000,
570 // 0.0000,
571 // 0.0000,
572 // 0.0000,
573 // 0.0000,
574 // 0.0000,
575 // 0.0000,
576 // 0.0000,
577 // 0.0000,
578 // 0.0000,
579 // 0.0000,
580 // 0.0000,
581 // 0.0000,
582 // 0.0000,
583 // 0.0000,
584 // 0.0000,
585 // 0.0000,
586 // 0.0000,
587 // 0.0001,
588 // 0.0001,
589 // 0.0002,
590 // 0.0003,
591 // 0.0006,
592 // 0.0009,
593 // 0.0014,
594 // 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), \
38         ULA_direction);
39
40     // the coefficients for the decimation filter
41     torch::Tensor lowpassfiltercoefficients =
42         torch::tensor({LOWPASS_DECIMATE_FILTER_COEFFICIENTS}).to(torch::kFloat);
43
44     // assigning values
45     ula_fls->num_sensors      = num_sensors;    // assigning number of sensors
46     ula_fls->inter_element_spacing = inter_element_spacing; // assigning inter-element spacing
47     ula_fls->coordinates      = ULA_coordinates; // assigning ULA coordinates
48     ula_fls->sampling_frequency = sampling_frequency; // assigning sampling frequencys
49     ula_fls->recording_period  = recording_period; // assigning recording period
50     ula_fls->sensorDirection   = ULA_direction; // ULA direction
51     ula_fls->lowpassFilterCoefficientsForDecimation = lowpassfiltercoefficients;
52
53     // assigning values
54     ula_port->num_sensors      = num_sensors;    // assigning number of sensors
55     ula_port->inter_element_spacing = inter_element_spacing; // assigning inter-element spacing
56     ula_port->coordinates      = ULA_coordinates; // assigning ULA coordinates
57     ula_port->sampling_frequency = sampling_frequency; // assigning sampling frequencys
58     ula_port->recording_period  = recording_period; // assigning recording period
59     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/vrsreeganesh/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: Convolving 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     // printing sampling frequency and bandwidth
126     std::cout << "main:: auv.transmitter_fls.bandwidth = " << auv.transmitter_fls.bandwidth << std::endl;
127     std::cout << "main:: auv.ULA_fls.sampling_frequency = " << auv.ULA_fls.sampling_frequency << std::endl;
128
129     // mimicking movement
130     int number_of_stophops = 1;
131     // if (true) return 0;
132     for(int i = 0; i<number_of_stophops; ++i){
133
134         // time measuring

```

```

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

```



```
204
205
206
207
208
209
210 // returning
211 return 0;
212 }
```

---

# Chapter 9

## Reading

### 9.1 Primary Books

- 1.

### 9.2 Interesting Papers