

Autonomous Underwater Vehicle: A Surveillance Protocol

S.V. Rajendran

February 3, 2025

Preface

This project is an attempt at combining all of my major skills into creating a truly sophisticated project real world project. The aim of this project is to come up with a perception and control pipeline for AUVs for maritime surveillance. As such, the work involves creating a number of sub-pipelines.

The first is the signal simulation and geometry pipeline. This pipeline takes care of creating the underwater profile and the signal simulation that is involved for the perception stack.

The perception stack for the AUV is one front-looking-SONAR and two side-scan SONARs. The parameters used for this project are obtained from that of NOAA ships that are publicly available. No proprietary parameters or specifications have been included as part of this project. The three SONARs help the AUV perceive the environment around it. The goal of the AUV is to essentially map the sea-floor and flag any new alien bodies in the “water”-space.

The control stack essentially assists in controlling the AUV in achieving the goal by controlling the AUV to spend minimal energy in achieving the goal of mapping. The terrains are randomly generated and thus, intelligent control is important to perceive the surrounding environment from the acoustic-images and control the AUV accordingly. The AUV is currently granted six degrees of freedom. The policy will be trained using a reinforcement learning approach (DQN is the plan). The aim is to learn a policy that will successfully learn how to achieve the goals of the AUV while also learning and adapting to the different kinds of terrains the first pipeline creates. To that end, this will be an online algorithm since the simulation cannot truly cover real terrains.

The project is currently written in C++. Despite the presence of significant deep learning aspects of the project, we choose C++ due to the real-time nature of the project and this is not merely a prototype. In addition, to enable the learning aspect, we use LibTorch (the C++ API to PyTorch).

Introduction

Contents

Preface	i
Introduction	ii
1 Setup	1
1.1 Overview	1
2 Underwater Environment Setup	2
2.1 Seafloor Setup	2
2.2 Additional Structures	2
3 Hardware Setup	3
3.1 Transmitter	3
3.2 Uniform Linear Array	3
3.3 Marine Vessel	3
4 Geometry	4
4.1 Ray Tracing	4
4.1.1 Pairwise Dot-Product	4
4.1.2 Range Histogram Method	4
5 Signal Simulation	6
5.1 Transmitted Signal	6
5.2 Signal Simulation	6
6 Imaging	8
7 Results	10
8 Software	11
8.1 Class Definitions	11
8.1.1 Class: Scatter	11
8.1.2 Class: Transmitter	13
8.1.3 Class: Uniform Linear Array	20
8.1.4 Class: Autonomous Underwater Vehicle	26
8.2 Setup Scripts	33
8.2.1 Seafloor Setup	33
8.2.2 Transmitter Setup	35

8.2.3	Uniform Linear Array	37
8.2.4	AUV Setup	38
8.3	Function Definitions	40
8.3.1	Cartesian Coordinates to Spherical Coordinates	40
9	Reading	41
9.1	Primary Books	41
9.2	Interesting Papers	41

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.

Chapter 2

Underwater Environment Setup

Overview

- The underwater environment is modelled using discrete scatterers.
- They contain two attributes: coordinates and reflectivity.

2.1 Seafloor Setup

- The sea-floor is the first set of scatterers we introduce.
- A simple flat or flat-ish mesh of scatterers.
- Further structures are simulated on top of this.
- The seafloor setup script is written in section 8.2.1;

2.2 Additional Structures

- We create additional scatters on the second layer.
- For now, we stick to simple spheres, boxes and so on;

Chapter 3

Hardware Setup

Overview

3.1 Transmitter

3.2 Uniform Linear Array

3.3 Marine Vessel

Chapter 4

Geometry

Overview

4.1 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.

4.1.1 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.

4.1.2 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 checkbox has not been filled. Since, for the filled ones, the filled scatter will shadow the others in the following range cells.

Algorithm 1 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

Signal Simulation

Overview

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

5.1 Transmitted Signal

- We use a linear frequency modulated signal.
- The signal is defined in setup-script of the transmitter. Please refer to section: 8.1.2;

5.2 Signal Simulation

1. First we obtain the set of scatterers that reflect the transmitted signal.
2. The distance between all the sensors and the scatterer distances are calculated.
3. The time of flight from the transmitter to each scatterer and each sensor is calculated.
4. This time is then calculated into sample number by multiplying with the sampling-frequency of the uniform linear arrays.
5. 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.
6. 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.

7. 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.

Chapter 6

Imaging

Overview

- Present different imaging methods.

Decimation

1. The signals received by the sensors have a huge number of samples in it. Storing that kind of information, especially when it will be accumulated over a long time like in the case of synthetic aperture SONAR, is impractical.
2. Since the transmitted signal is LFM and non-baseband, this means that making the signal a complex baseband and decimating it will result in smaller data but same information.
3. So what we do is once we receive the signal at a stop-hop, we baseband the signal, low-pass filter it around the bandwidth and then decimate the signal. This reduces the sample number by a lot.
4. Since we're working with spotlight-SAS, this can be further reduced by beamforming the received signals in the direction of the patch and just storing the single beam. (This needs validation from Hareesh sir btw)

Match-Filtering

- A match-filter is any signal, that which when multiplied with another signal produces a signal that has a flat frequency-response = an impulse basically. (I might've butchered that definition but this will be updated later)
- This is created by time-reversing and calculating the complex conjugate of the signal.
- The resulting match-filter is then convolved with the received signal. This will result in a sincs being placed where impulse responses would've been if we used an infinite bandwidth signal.

Questions

- Do we match-filter before beamforming or after. I do realize that theoretically they're the same but practically, does one conserve resolution more than the other.

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         PRINTSMALLLINE
67
68         // returning os
69         return os;
70     }
71 }
72 };

```

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 steps:
194 1. Find azimuth and range of all points.
195 2. Find azimuth and range of current pointing vector.
196 3. Subtract azimuth and range of points from that of azimuth and range of current pointing vector
197 4. Use tilted ellipse equation to find points in the ellipse
198 -----*/
199 void subsetScatterers(ScattererClass* scatterers,
200 float tilt_angle){
201
202     // translationally change origin
203     scatterers->coordinates = scatterers->coordinates - this->location; if(DEBUGMODE_TRANSMITTER)
204     std::cout<<"\t\t TransmitterClass: line 188 "<<std::endl;
205
206     /*
207     Note: I think something we can do is see if we can subset the matrices by checking coordinate values
208     right away. If one of the coordinate values is x (relative coordinates), we know for sure that
209     the distance is greater than x, for sure. So, maybe that's something that we can work with

```

```

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

```

```

262 torch::Tensor spherical_coordinates = fCart2Sph(scatterers->coordinates); std::cout<<"\t\t
    TransmitterClass: line 252 "<<std::endl;
263
264 // finding maximum range
265 torch::Tensor maxdistanceofpoints = torch::max(spherical_coordinates[2]); std::cout<<"\t\t
    TransmitterClass: line 256 "<<std::endl;
266
267 // calculating number of range-cells (verified)
268 int numrangecells = std::ceil(maxdistanceofpoints.item<int>()/this->rangeQuantSize);
269
270 // finding range-cell boundaries (verified)
271 torch::Tensor rangeBoundaries = \
272     torch::linspace(this->rangeQuantSize, \
273         numrangecells * this->rangeQuantSize, \
274         numrangecells); std::cout<<"\t\t TransmitterClass: line 263 "<<std::endl;
275
276 // creating the checkbox (verified)
277 int numazimuthcells = std::ceil(this->azimuthal_beamwidth * this->azimuthQuantDensity);
278 int numelevationcells = std::ceil(this->elevation_beamwidth * this->elevationQuantDensity);
    std::cout<<"\t\t TransmitterClass: line 267 "<<std::endl;
279
280 // finding the deltas
281 float delta_azimuth = this->azimuthal_beamwidth / numazimuthcells;
282 float delta_elevation = this->elevation_beamwidth / numelevationcells; std::cout<<"\t\t
    TransmitterClass: line 271"<<std::endl;
283
284 // creating the centers (verified)
285 torch::Tensor azimuth_centers = torch::linspace(delta_azimuth/2, \
286     numazimuthcells * delta_azimuth - delta_azimuth/2, \
287     numazimuthcells);
288 torch::Tensor elevation_centers = torch::linspace(delta_elevation/2, \
289     numelevationcells * delta_elevation - delta_elevation/2, \
290     numelevationcells); std::cout<<"\t\t TransmitterClass:
    line 279"<<std::endl;
291
292 // centering (verified)
293 azimuth_centers = azimuth_centers + torch::tensor({this->azimuthal_angle - \
294     (this->azimuthal_beamwidth/2)}).to(torch::kFloat);
295 elevation_centers = elevation_centers + torch::tensor({this->elevation_angle - \
296     (this->elevation_beamwidth/2)}).to(torch::kFloat);
    std::cout<<"\t\t TransmitterClass: line
    285"<<std::endl;
297
298 // building checkboxes
299 torch::Tensor checkbox = torch::zeros({numelevationcells, numazimuthcells}, torch::kBool);
300 torch::Tensor finalScatterBox = torch::zeros({numelevationcells, numazimuthcells,
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\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(), \
414     index_where_min_range_is});
415 torch::Tensor closest_reflectivity = reflectivity_along_aziele.index({torch::indexing::Slice(),
    \
416     index_where_min_range_is});
417
418 // filling the matrices up
419 finalScatterBox.index_put_({ele_index, azi_index, torch::indexing::Slice()}, \
420     closest_coord.reshape({1,1,3}));
421 finalReflectivityBox.index_put_({ele_index, azi_index}, \
422     closest_reflectivity);
423 checkbox.index_put_({ele_index, azi_index}, \
424     true);
425
426 }
427 }
428 }
429
430
431
432
433 };

```

8.1.3 Class: Uniform Linear Array

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

```

1  // bringing in the header files
2  #include <cstdint>
3  #include <iostream>
4  #include <stdexcept>
5  #include <torch/torch.h>
6
7
8  // bringing in the functions
9  #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fPrintTensorSize.cpp"
10 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fConvolveColumns.cpp"
11 #include "ScattererClass.h"
12 #include "TransmitterClass.h"
13 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fBuffer2D.cpp"
14
15 #pragma once
16
17 // hash defines
18 #ifndef PRINTSPACE
19     #define PRINTSPACE    std::cout<<"\n\n\n\n\n\n\n\n\n\n"<<std::endl;
20 #endif
21 #ifndef PRINTSMALLLINE
22     #define PRINTSMALLLINE std::cout<<"-----"<<std::endl;
23 #endif
24 #ifndef PRINTLINE
25     #define PRINTLINE      std::cout<<"===== "<<std::endl;
26 #endif
27
28 #ifndef DEVICE
29     // #define DEVICE      torch::kMPS
30     #define DEVICE      torch::kCPU
31 #endif
32
33 #define PI              3.14159265
34
35 // #define DEBUG_ULA true
36 #define DEBUG_ULA false
37
38
39
40 class ULAClass{
41 public:
42     // intrinsic parameters
43     int num_sensors;           // number of sensors
44     float inter_element_spacing; // space between sensors
45     torch::Tensor coordinates; // coordinates of each sensor
46     float sampling_frequency;   // sampling frequency of the sensors
47     float recording_period;     // recording period of the ULA
48     torch::Tensor location;     // location of first coordinate
49
50     // derived stuff
51     torch::Tensor sensorDirection;
52     torch::Tensor signalMatrix;
53
54     // decimation-related
55     int decimation_factor;
56     torch::Tensor lowpassFilterCoefficientsForDecimation; //
57
58     // imaging related
59     float range_resolution; // theoretical range-resolution =  $\frac{c}{2B}$ 
60     float azimuthal_resolution; // theoretical azimuth-resolution =  $\frac{\lambda}{(N-1)*inter-element-distance}$ 
61
62     float range_cell_size; // the range-cell quanta we're choosing for efficiency trade-off
63     float azimuth_cell_size; // the azimuth quanta we're choosing
64
65     // constructor

```

```

66 ULAClass(int numsensors          = 32,
67          float inter_element_spacing = 1e-3,
68          torch::Tensor coordinates = torch::zeros({3, 2}),
69          float sampling_frequency = 48e3,
70          float recording_period    = 1,
71          torch::Tensor location    = torch::zeros({3,1}),
72          torch::Tensor signalMatrix = torch::zeros({1, 32}),
73          torch::Tensor lowpassFilterCoefficientsForDecimation = torch::zeros({1,10})):
74     num_sensors(numsensors),
75     inter_element_spacing(inter_element_spacing),
76     coordinates(coordinates),
77     sampling_frequency(sampling_frequency),
78     recording_period(recording_period),
79     location(location),
80     signalMatrix(signalMatrix),
81     lowpassFilterCoefficientsForDecimation(lowpassFilterCoefficientsForDecimation){
82         // calculating ULA direction
83         torch::Tensor sensorDirection = coordinates.slice(1, 0, 1) - coordinates.slice(1, 1, 2);
84
85         // normalizing
86         float normvalue = torch::linalg_norm(sensorDirection, 2, 0, true, torch::kFloat).item<float>();
87         if (normvalue != 0){
88             sensorDirection = sensorDirection / normvalue;
89         }
90
91         // copying direction
92         this->sensorDirection = sensorDirection;
93     }
94
95     // overriding printing
96     friend std::ostream& operator<<(std::ostream& os, ULAClass& ula){
97         os<<"\t number of sensors : "<<ula.num_sensors <<std::endl;
98         os<<"\t inter-element spacing: "<<ula.inter_element_spacing <<std::endl;
99         os<<"\t sensor-direction " <<torch::transpose(ula.sensorDirection, 0, 1)<<std::endl;
100         PRINTSMALLLINE
101         return os;
102     }
103
104     /* =====
105     Aim: Init
106     ===== */
107     void init(){
108         ;
109     }
110
111     // overloading the "=" operator
112     ULAClass& operator=(const ULAClass& other){
113         // checking if copying to the same object
114         if(this == &other){
115             return *this;
116         }
117
118         // copying everything
119         this->num_sensors          = other.num_sensors;
120         this->inter_element_spacing = other.inter_element_spacing;
121         this->coordinates          = other.coordinates.clone();
122         this->sampling_frequency    = other.sampling_frequency;
123         this->recording_period      = other.recording_period;
124         this->sensorDirection       = other.sensorDirection.clone();
125
126         // new additions
127         // this->location            = other.location;
128         this->lowpassFilterCoefficientsForDecimation = other.lowpassFilterCoefficientsForDecimation;
129         // this->sensorDirection     = other.sensorDirection.clone();
130         // this->signalMatrix        = other.signalMatrix.clone();
131
132
133         // returning
134         return *this;
135     }
136
137     // build sensor-coordinates based on location
138     void buildCoordinatesBasedOnLocation(){

```

```

139
140 // length-normalize the sensor-direction
141 this->sensorDirection = torch::div(this->sensorDirection, torch::linalg_norm(this->sensorDirection, \
142                                     2, 0, true, \
143                                     torch::kFloat));
144
145 if(DEBUG_ULA) std::cout<<"\t ULAClass: line 105 \n";
146
147 // multiply with inter-element distance
148 this->sensorDirection = this->sensorDirection * this->inter_element_spacing;
149 this->sensorDirection = this->sensorDirection.reshape({this->sensorDirection.numel(), 1});
150 if(DEBUG_ULA) std::cout<<"\t ULAClass: line 110 \n";
151
152 // create integer-array
153 // torch::Tensor integer_array = torch::linspace(0, \
154 //                                     this->num_sensors-1, \
155 //                                     this->num_sensors).reshape({1,
156 //                                     this->num_sensors}).to(torch::kFloat);
157 torch::Tensor integer_array = torch::linspace(0, \
158 //                                     this->num_sensors-1, \
159 //                                     this->num_sensors).reshape({1, this->num_sensors});
160 std::cout<<"integer_array = "; fPrintTensorSize(integer_array);
161 if(DEBUG_ULA) std::cout<<"\t ULAClass: line 116 \n";
162
163 // this->coordinates = torch::mul(torch::tile(integer_array, {3, 1}).to(torch::kFloat), \
164 //                                     torch::tile(this->sensorDirection, {1,
165 //                                     this->num_sensors}).to(torch::kFloat));
166 torch::Tensor test = torch::mul(torch::tile(integer_array, {3, 1}).to(torch::kFloat), \
167 //                                     torch::tile(this->sensorDirection, {1,
168 //                                     this->num_sensors}).to(torch::kFloat));
169 this->coordinates = this->location + test;
170 if(DEBUG_ULA) std::cout<<"\t ULAClass: line 120 \n";
171
172 }
173
174 // signal simulation for the current sensor-array
175 void nfdc_simulateSignal(ScattererClass* scatterers,
176                         TransmitterClass* transmitterObj){
177
178     // creating signal matrix
179     int numsamples = std::ceil((this->sampling_frequency * this->recording_period));
180     this->signalMatrix = torch::zeros({numsamples, this->num_sensors}).to(torch::kFloat);
181
182     // getting shape of coordinates
183     std::vector<int64_t> scatterers_coordinates_shape = scatterers->coordinates.sizes().vec();
184
185     // making a slot out of the coordinates
186     torch::Tensor slottedCoordinates = \
187         torch::permute(scatterers->coordinates.reshape({scatterers_coordinates_shape[0], \
188 //                                     scatterers_coordinates_shape[1], \
189 //                                     1
190 //                                     }, \
191 //                                     {2, 1, 0}).reshape({1, (int)(scatterers->coordinates.numel()/3), 3}));
192
193     // repeating along the y-direction number of sensor times.
194     slottedCoordinates = torch::tile(slottedCoordinates, {this->num_sensors, 1, 1});
195     std::vector<int64_t> slottedCoordinates_shape = slottedCoordinates.sizes().vec();
196
197     // finding the shape of the sensor-coordinates
198     std::vector<int64_t> sensor_coordinates_shape = this->coordinates.sizes().vec();
199
200     // creating a slot tensor out of the sensor-coordinates
201     torch::Tensor slottedSensors = \
202         torch::permute(this->coordinates.reshape({sensor_coordinates_shape[0], \
203 //                                     sensor_coordinates_shape[1], \
204 //                                     1
205 //                                     }, {2, 1, 0}).reshape({(int)(this->coordinates.numel()/3),
206 //                                     \
207 //                                     1, \
208 //                                     3}));
209
210     // repeating slices along the y-coordinates
211     slottedSensors = torch::tile(slottedSensors, {1, slottedCoordinates_shape[1], 1});
212
213     // slotting the coordinate of the transmitter

```

```

208 torch::Tensor slotted_location = torch::permute(this->location.reshape({3, 1, 1}), \
209         {2, 1, 0}).reshape({1,1,3});
210 slotted_location = torch::tile(slotted_location, \
211     {slottedCoordinates_shape[0], slottedCoordinates_shape[1], 1});
212
213 // subtracting to find the relative distances
214 torch::Tensor distBetweenScatterersAndSensors = \
215     torch::linalg_norm(slottedCoordinates - slottedSensors, 2, 2, true, torch::kFloat);
216
217 // subtracting distance between relative fields
218 torch::Tensor distBetweenScatterersAndTransmitter = \
219     torch::linalg_norm(slottedCoordinates - slotted_location, 2, 2, true, torch::kFloat);
220
221 // adding up the distances
222 torch::Tensor distOfFlight = distBetweenScatterersAndSensors + distBetweenScatterersAndTransmitter;
223 torch::Tensor timeOfFlight = distOfFlight/1500;
224 torch::Tensor samplesOfFlight = torch::floor(timeOfFlight.squeeze() * this->sampling_frequency);
225
226 // Adding pulses
227 for(int sensor_index = 0; sensor_index < this->num_sensors; ++sensor_index){
228     for(int scatter_index = 0; scatter_index < samplesOfFlight[0].numel(); ++scatter_index){
229
230         // getting the sample where the current scatter's contribution must be placed.
231         int where_to_place = samplesOfFlight.index({sensor_index, scatter_index}).item<int>();
232
233         // checking whether that point is out of bounds
234         if(where_to_place >= numsamples) continue;
235
236         // placing a point in there
237         this->signalMatrix.index_put_({where_to_place, sensor_index}, \
238             this->signalMatrix.index({where_to_place, sensor_index}) + \
239             torch::tensor({1}).to(torch::kFloat));
240
241         this->signalMatrix.index_put_({where_to_place, sensor_index}, \
242             this->signalMatrix.index({where_to_place, sensor_index}) + \
243             scatterers->reflectivity.index({0, scatter_index}) );
244     }
245 }
246
247 // Convolving signals with transmitted signal
248 torch::Tensor signalTensorAsArgument = \
249     transmitterObj->Signal.reshape({transmitterObj->Signal.numel(),1});
250 signalTensorAsArgument = torch::tile(signalTensorAsArgument, \
251     {1, this->signalMatrix.size(1)});
252
253 // convolving the pulse-matrix with the signal matrix
254 fConvolveColumns(this->signalMatrix, \
255     signalTensorAsArgument);
256
257 // trimming the convolved signal since the signal matrix length remains the same
258 this->signalMatrix = this->signalMatrix.index({torch::indexing::Slice(0, numsamples), \
259     torch::indexing::Slice()});
260
261 // printing the shape
262 if(DEBUG_ULA) {std::cout<<"\t\t\t\t this->signalMatrix.shape (after signal sim) = ";
    fPrintTensorSize(this->signalMatrix);}
263
264 return;
265 }
266
267 // decimating the obtained signal
268 void nfdc_decimateSignal(TransmitterClass* transmitterObj){
269
270     if(DEBUG_ULA) std::cout<<"\t\t\t\t ULAClass::nfdc_decimateSignal: entered "<<std::endl;
271
272     // basebanding the signal
273     torch::Tensor integerArray = torch::linspace(0, \
274         this->signalMatrix.size(0)-1, \
275         this->signalMatrix.size(0)).reshape({this->signalMatrix.size(0),
276             1}); if(DEBUG_ULA) std::cout<<"\t\t\t\t
277         ULAClass::nfdc_decimateSignal: 247"<<std::endl;
276
277     if(DEBUG_ULA) std::cout<<"integerArray.shape = "<<integerArray.sizes().vec()<<std::endl;
    integerArray
        = torch::tile(integerArray, \

```

```

278         {1, this->num_sensors}); if(DEBUG_ULA) std::cout<<"\t\t\t
                ULAClass::nfdc_decimateSignal: 249"<<std::endl;
279 if(DEBUG_ULA) std::cout<<"integerArray.shape = "<<integerArray.sizes().vec()<<std::endl;
280 integerArray
281     = torch::exp(torch::complex(torch::zeros({1}), torch::ones({1}))) * \
282         transmitterObj->fc * \
                integerArray); if(DEBUG_ULA) std::cout<<"\t\t\t
                ULAClass::nfdc_decimateSignal: 250"<<std::endl;

283
284 if(DEBUG_ULA) std::cout<<"this->signalMatrix.shape = "<<this->signalMatrix.sizes().vec()<<std::endl;
285 if(DEBUG_ULA) std::cout<<"integerArray.shape = "<<integerArray.sizes().vec()<<std::endl;
286 this->signalMatrix = torch::mul(this->signalMatrix, \
287     integerArray); if(DEBUG_ULA) std::cout<<"\t\t\t
                ULAClass::nfdc_decimateSignal: 254"<<std::endl;

288
289 // low-pass filter
290 torch::Tensor lowpassfilter_impulseresponse = \
291     this->lowpassFilterCoefficientsForDecimation.reshape({this->lowpassFilterCoefficientsForDecimation.numel(),
        1}); if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_decimateSignal: 263"<<std::endl;
292 lowpassfilter_impulseresponse = torch::tile(lowpassfilter_impulseresponse, \
293     {1, this->signalMatrix.size(1)}); if(DEBUG_ULA) std::cout<<"\t\t\t
                ULAClass::nfdc_decimateSignal: 260"<<std::endl;

294
295 // Convolving
296 fConvolveColumns(this->signalMatrix, lowpassfilter_impulseresponse); if(DEBUG_ULA) std::cout<<"\t\t\t
                ULAClass::nfdc_decimateSignal: 263"<<std::endl;

297
298 // downsampling
299 int decimation_factor = std::floor(this->sampling_frequency/transmitterObj->bandwidth); if(DEBUG_ULA)
        std::cout<<"\t\t\t ULAClass::nfdc_decimateSignal: 284"<<std::endl;
300 this->decimation_factor = decimation_factor;
301 if(DEBUG_ULA) std::cout<<"\t\t\t\t\t decimation_factor = "<<decimation_factor<<std::endl;
302 int numsamples_after_decimation = std::floor(this->signalMatrix.size(0)/decimation_factor);
        if(DEBUG_ULA) std::cout<<"\t\t\t\t\t ULAClass::nfdc_decimateSignal: 285"<<std::endl;
303 torch::Tensor samplingIndices = \
304     torch::linspace(0, \
305         numsamples_after_decimation * decimation_factor, \
306         numsamples_after_decimation).to(torch::kInt); if(DEBUG_ULA) std::cout<<"\t\t\t
                ULAClass::nfdc_decimateSignal: 289"<<std::endl;

307
308 this->signalMatrix = \
309     this->signalMatrix.index({samplingIndices, torch::indexing::Slice()}); if(DEBUG_ULA)
310     std::cout<<"\t\t\t\t\t ULAClass::nfdc_decimateSignal: 273"<<std::endl;

311
312 if(DEBUG_ULA) std::cout<<"this->signalMatrix.shape = "<<this->signalMatrix.sizes().vec()<<std::endl;
313
314 }
315
316 /* =====
317 Aim: precompute delay-matrices
318 ===== */
319 void nfdc_precomputeDelayMatrices(TransmitterClass* transmitterObj){
320
321     // calculating range-related parameters
322     this->range_resolution = 1500/(2 * transmitterObj->fc);
323     this->range_cell_size = 40 * this->range_resolution;
324     int number_of_range_cells = ((this->recording_period * 1500)/2)/this->range_cell_size;
325
326     // status printing
327     if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass: this->range_resolution = " << this->range_resolution <<
        std::endl;
328     if(DEBUG_ULA) std::cout<<"\t\t\t\t\t ULAClass: this->range_cell_size = " << this->range_cell_size <<
        std::endl;
329     if(DEBUG_ULA) std::cout<<"\t\t\t\t\t ULAClass: number_of_range_cells = " << number_of_range_cells <<
        std::endl;

330
331     // calculating azimuth-related parameters
332     this->azimuthal_resolution =
        (transmitterObj->fc)/((this->num_sensors-1)*this->inter_element_spacing);
333     this->azimuth_cell_size = 2 * this->azimuthal_resolution;
334     int number_of_azimuths_to_image = std::ceil(transmitterObj->azimuthal_beamwidth /
        this->azimuth_cell_size);
335

```

```

336 // printing
337 if(DEBUG_ULA) std::cout << "\t\t\t ULAClass: this->azimuthal_resolution = " <<
    this->azimuthal_resolution << std::endl;
338 if(DEBUG_ULA) std::cout << "\t\t\t ULAClass: this->azimuth_cell_size = " << this->azimuth_cell_size
    << std::endl;
339 if(DEBUG_ULA) std::cout << "\t\t\t ULAClass: number_of_azimuths_to_image = " <<
    number_of_azimuths_to_image << std::endl;
340
341 // find the centers of the range-cells.
342 torch::Tensor range_centers = \
343     torch::linspace(this->azimuth_cell_size/2, \
344         (number_of_range_cells - 0.5)*this->azimuth_cell_size, \
345         number_of_range_cells);
346
347 // finding the centers of azimuth-cells
348 torch::Tensor azimuth_centers = \
349     torch::linspace(this->range_cell_size/2, \
350         (number_of_azimuths_to_image - 0.5) * this->range_cell_size, \
351         number_of_azimuths_to_image);
352 }
353
354 // beamforming the signal
355 void nfdc_beamforming(TransmitterClass* transmitterObj){
356
357     // buffering the 2D signal
358     if(this->signalMatrix.size(1) != this->num_sensors)
359         throw std::runtime_error("The second dimension doesn't correspond to the number of sensors \n");
360
361     // calculating the frame-size
362     int frame_size = std::ceil((2 * this->range_cell_size/1500) * \
363         (this->sampling_frequency / this->decimation_factor));
364
365     // printing
366     if(DEBUG_ULA) std::cout<<"\t\t\t this->range_cell_size = " << this->range_cell_size << std::endl;
367     if(DEBUG_ULA) std::cout<<"\t\t\t this->sampling_frequency = " << this->sampling_frequency << std::endl;
368     if(DEBUG_ULA) std::cout<<"\t\t\t this->decimation_factor = " << this->decimation_factor << std::endl;
369     if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass: frame_size = " << frame_size << std::endl;
370
371     // adding the batch-dimension
372     if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass: this->signalMatrix.sizes().vec() (before) = "
        "<<this->signalMatrix.sizes().vec()<<std::endl;
373     this->signalMatrix = \
374         this->signalMatrix.reshape({1, \
375             this->signalMatrix.size(0), \
376             this->signalMatrix.size(1)});
377     if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass: this->signalMatrix.sizes().vec() (before) = "
        "<<this->signalMatrix.sizes().vec()<<std::endl;
378
379     // breaking apart the signal
380     fBuffer2D(this->signalMatrix, frame_size);
381     if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass: this->signalMatrix.sizes().vec() (after)= "
        "<<this->signalMatrix.sizes().vec()<<std::endl;
382
383
384
385
386
387
388 // creating a range-angle mesh for this
389
390 // find the different angles for which we're beamforming.
391
392 // find the delays for the different range-angle combinations
393
394 // splitting the signals into the different range-cells
395
396 // loop for beamforming all of em
397 }
398
399
400
401
402

```

```

403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419 };

```

8.1.4 Class: Autonomous Underwater Vehicle

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

```

1  #include "ScattererClass.h"
2  #include "TransmitterClass.h"
3  #include "ULAClass.h"
4  #include <iostream>
5  #include <ostream>
6  #include <torch/torch.h>
7  #include <cmath>
8
9
10 // // including functions
11 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fGetCurrentTimeFormatted.cpp"
12
13 #pragma once
14
15 // function to plot the thing
16 void fPlotTensors(){
17     system("python /Users/vrsreeganesh/Documents/GitHub/AUV/Code/Python/TestingSaved_tensors.py");
18 }
19
20
21 void fSaveSeafloorScatteres(ScattererClass scatterer, \
22                             ScattererClass scatterer_fls, \
23                             ScattererClass scatterer_port, \
24                             ScattererClass scatterer_starboard){
25
26     // saving the ground-truth
27     ScattererClass SeafloorScatter_gt = scatterer;
28     torch::save(SeafloorScatter_gt.coordinates,
29                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt.pt");
30     torch::save(SeafloorScatter_gt.reflectivity,
31                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt_reflectivity.pt");
32
33     // saving coordinates
34     torch::save(scatterer_fls.coordinates,
35                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_fls_coordinates.pt");
36     torch::save(scatterer_port.coordinates,
37                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_port_coordinates.pt");
38     torch::save(scatterer_starboard.coordinates,
39                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_starboard_coordinates.pt");
40
41     // saving reflectivities
42     torch::save(scatterer_fls.reflectivity,
43                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_fls_coordinates_reflectivity.pt");
44     torch::save(scatterer_port.reflectivity,
45                 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_port_coordinates_reflectivity.pt");
46     torch::save(scatterer_starboard.reflectivity,

```



```

40     "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_starboard.coordinates_reflectivity.pt");
41 // plotting tensors
42 fPlotTensors();
43
44 // // saving the tensors
45 // if (true) {
46
47 //     // getting time ID
48 //     auto timeID = fGetCurrentTimeFormatted();
49
50 //     std::cout<<"\t\t\t\t\t Saving Tensors (timeID: "<<timeID<<)"<<std::endl;
51
52 //     // saving the ground-truth
53 //     ScattererClass SeafloorScatter_gt = scatterer;
54 //     torch::save(SeafloorScatter_gt.coordinates, \
55 //         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt.pt");
56 //     torch::save(SeafloorScatter_gt.reflectivity, \
57 //         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt_reflectivity.pt");
58
59
60 //     // saving coordinates
61 //     torch::save(scatterer_fls.coordinates, \
62 //         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_fls_coordinates.pt");
63 //     torch::save(scatterer_port.coordinates, \
64 //         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_port_coordinates.pt");
65 //     torch::save(scatterer_starboard.coordinates, \
66 //         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_starboard.coordinates.pt");
67
68 //     // saving reflectivities
69 //     torch::save(scatterer_fls.reflectivity, \
70 //         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_fls_coordinates_reflectivity.pt");
71 //     torch::save(scatterer_port.reflectivity, \
72 //         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_port_coordinates_reflectivity.pt");
73 //     torch::save(scatterer_starboard.reflectivity, \
74 //         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_starboard.coordinates_reflectivity.pt");
75
76 //     // plotting tensors
77 //     fPlotTensors();
78
79 //     // indicating end of thread
80 //     std::cout<<"\t\t\t\t\t Ended (timeID: "<<timeID<<)"<<std::endl;
81 // }
82 }
83
84 // including class-definitions
85 #include "Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/ScattererClass.h"
86
87 // hash defines
88 #ifndef PRINTSPACE
89 #define PRINTSPACE    std::cout<<"\n\n\n\n\n\n\n"<<std::endl;
90 #endif
91 #ifndef PRINTSMALLLINE
92 #define PRINTSMALLLINE std::cout<<"-----"<<std::endl;
93 #endif
94 #ifndef PRINTLINE
95 #define PRINTLINE    std::cout<<"===== "<<std::endl;
96 #endif
97
98 #ifndef DEVICE
99 #define DEVICE        torch::kMPS
100 // #define DEVICE      torch::kCPU
101 #endif
102
103 #define PI            3.14159265
104 // #define DEBUGMODE AUV true

```



```

105 #define DEBUGMODE_AUV false
106
107
108 class AUVClass{
109 public:
110     // Intrinsic attributes
111     torch::Tensor location;           // location of vessel
112     torch::Tensor velocity;           // current speed of the vessel [a vector]
113     torch::Tensor acceleration;       // current acceleration of vessel [a vector]
114     torch::Tensor pointing_direction; // direction to which the AUV is pointed
115
116     // uniform linear-arrays
117     ULAClass ULA_fls;                 // front-looking SONAR ULA
118     ULAClass ULA_port;                // mounted ULA [object of class, ULAClass]
119     ULAClass ULA_starboard;           // mounted ULA [object of class, ULAClass]
120
121     // transmitters
122     TransmitterClass transmitter_fls; // transmitter for front-looking SONAR
123     TransmitterClass transmitter_port; // mounted transmitter [obj of class, TransmitterClass]
124     TransmitterClass transmitter_starboard; // mounted transmitter [obj of class, TransmitterClass]
125
126     // derived or dependent attributes
127     torch::Tensor signalMatrix_1;     // matrix containing the signals obtained from ULA_1
128     torch::Tensor largeSignalMatrix_1; // matrix holding signal of synthetic aperture
129     torch::Tensor beamformedLargeSignalMatrix; // each column is the beamformed signal at each stop-hop
130
131     // plotting mode
132     bool plottingmode; // to suppress plotting associated with classes
133
134     // spotlight mode related
135     torch::Tensor absolute_coords_patch_cart; // cartesian coordinates of patch
136
137     // Synthetic Aperture Related
138     torch::Tensor ApertureSensorLocations; // sensor locations of aperture
139
140
141     /*=====
142     Aim: stepping motion
143     -----*/
144     void step(float timestep){
145
146         // updating location
147         this->location = this->location + this->velocity * timestep;
148         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 81 \n";
149
150         // updating attributes of members
151         this->syncComponentAttributes();
152         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 85 \n";
153     }
154
155     /*=====
156     Aim: updateAttributes
157     -----*/
158     void syncComponentAttributes(){
159
160         // updating ULA attributes
161         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 97 \n";
162
163         // updating locations
164         this->ULA_fls.location = this->location;
165         this->ULA_port.location = this->location;
166         this->ULA_starboard.location = this->location;
167
168         // updating the pointing direction of the ULAs
169         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 99 \n";
170         torch::Tensor ula_fls_sensor_direction_spherical = fCart2Sph(this->pointing_direction); //
            spherical coords
171         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 101 \n";
172         ula_fls_sensor_direction_spherical[0] = ula_fls_sensor_direction_spherical[0] - 90;
173         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 98 \n";
174         torch::Tensor ula_fls_sensor_direction_cart = fSph2Cart(ula_fls_sensor_direction_spherical);
175         if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 100 \n";
176

```

```

177     this->ULA_fls.sensorDirection      = ula_fls_sensor_direction_cart; // assigning sensor directionf or
178     ULA-FLS
179     this->ULA_port.sensorDirection      = -this->pointing_direction;    // assigning sensor direction for
180     ULA-Port
181     this->ULA_starboard.sensorDirection = -this->pointing_direction;    // assigning sensor direction for
182     ULA-Starboard
183     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 105 \n";
184
185     // // calling the function to update the arguments
186     // this->ULA_fls.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line
187     109 \n";
188     // this->ULA_port.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line
189     111 \n";
190     // this->ULA_starboard.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) std::cout<<"\t AUVClass:
191     line 113 \n";
192
193     // updating transmitter locations
194     this->transmitter_fls.location      = this->location;
195     this->transmitter_port.location     = this->location;
196     this->transmitter_starboard.location = this->location;
197     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 102 \n";
198
199     // updating transmitter pointing directions
200     this->transmitter_fls.updatePointingAngle( this->pointing_direction);
201     this->transmitter_port.updatePointingAngle( this->pointing_direction);
202     this->transmitter_starboard.updatePointingAngle( this->pointing_direction);
203     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 108 \n";
204 }
205
206 /*=====
207 Aim: operator overriding for printing
208 -----*/
209 friend std::ostream& operator<<(std::ostream& os, AUVClass &auv){
210     os<<"\t location = "<<torch::transpose(auv.location, 0, 1)<<std::endl;
211     os<<"\t velocity = "<<torch::transpose(auv.velocity, 0, 1)<<std::endl;
212     return os;
213 }
214
215 /*=====
216 Aim: Subsetting Scatterers
217 -----*/
218 void subsetScatterers(ScattererClass* scatterers,\
219     TransmitterClass* transmitterObj,\
220     float tilt_angle){
221
222     // ensuring components are synced
223     this->syncComponentAttributes();
224     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 120 \n";
225
226     // calling the method associated with the transmitter
227     if(DEBUGMODE_AUV) {std::cout<<"\t\t scatterers.shape = "; fPrintTensorSize(scatterers->coordinates);}
228     if(DEBUGMODE_AUV) std::cout<<"\t\t tilt_angle = "<<tilt_angle<<std::endl;
229     transmitterObj->subsetScatterers(scatterers, tilt_angle);
230     if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 124 \n";
231 }
232
233 // pitch-correction matrix
234 torch::Tensor createYawCorrectionMatrix(torch::Tensor pointing_direction_spherical, \
235     float target_azimuth_deg){
236
237     // building parameters
238     torch::Tensor azimuth_correction      =
239         torch::tensor({target_azimuth_deg}).to(torch::kFloat).to(DEVICE) - \
240         pointing_direction_spherical[0];
241     torch::Tensor azimuth_correction_radians = azimuth_correction * PI / 180;
242
243     torch::Tensor yawCorrectionMatrix = \
244         torch::tensor({torch::cos(azimuth_correction_radians).item<float>(), \
245             torch::cos(torch::tensor({90}).to(torch::kFloat).to(DEVICE)*PI/180 +
246                 azimuth_correction_radians).item<float>(), \
247             (float)0, \
248             torch::sin(azimuth_correction_radians).item<float>(), \

```

```

242         torch::sin(torch::tensor({90}).to(torch::kFloat).to(DEVICE)*PI/180 +
                azimuth_correction_radians).item<float>(), \
243         (float)0, \
244         (float)0, \
245         (float)0, \
246         (float)1}).reshape({3,3}).to(torch::kFloat).to(DEVICE);
247
248     // returning the matrix
249     return yawCorrectionMatrix;
250 }
251
252 // pitch-correction matrix
253 torch::Tensor createPitchCorrectionMatrix(torch::Tensor pointing_direction_spherical, \
254         float target_elevation_deg){
255
256     // building parameters
257     torch::Tensor elevation_correction =
258         torch::tensor({target_elevation_deg}).to(torch::kFloat).to(DEVICE) - \
259         pointing_direction_spherical[1];
260     torch::Tensor elevation_correction_radians = elevation_correction * PI / 180;
261
262     // creating the matrix
263     torch::Tensor pitchCorrectionMatrix = \
264         torch::tensor({(float)1, \
265             (float)0, \
266             (float)0, \
267             (float)0, \
268             torch::cos(elevation_correction_radians).item<float>(), \
269             torch::cos(torch::tensor({90}).to(torch::kFloat).to(DEVICE)*PI/180 +
                elevation_correction_radians).item<float>(), \
270             (float)0, \
271             torch::sin(elevation_correction_radians).item<float>(), \
272             torch::sin(torch::tensor({90}).to(torch::kFloat).to(DEVICE)*PI/180 +
                elevation_correction_radians).item<float>())}).reshape({3,3}).to(torch::kFloat);
273
274     // returning the matrix
275     return pitchCorrectionMatrix;
276 }
277
278 // Signal Simulation
279 void simulateSignal(ScattererClass& scatterer){
280
281     // making three copies
282     ScattererClass scatterer_fls = scatterer;
283     ScattererClass scatterer_port = scatterer;
284     ScattererClass scatterer_starboard = scatterer;
285
286     // printing size of scatterers before subsetting
287     std::cout<< " > AUVClass: Beginning Scatterer Subsetting"<<std::endl;
288     std::cout<<"\t AUVClass: scatterer_fls.coordinates.shape (before) = ";
289     fPrintTensorSize(scatterer_fls.coordinates);
290     std::cout<<"\t AUVClass: scatterer_port.coordinates.shape (before) = ";
291     fPrintTensorSize(scatterer_port.coordinates);
292     std::cout<<"\t AUVClass: scatterer_starboard.coordinates.shape (before) = ";
293     fPrintTensorSize(scatterer_starboard.coordinates);
294
295     // finding the pointing direction in spherical
296     torch::Tensor auv_pointing_direction_spherical = fCart2Sph(this->pointing_direction);
297
298     // asking the transmitters to subset the scatterers by multithreading
299     std::thread transmitterFLSSubset_t(&AUVClass::subsetScatterers, this, \
300         &scatterer_fls, \
301         &this->transmitter_fls, \
302         (float)0);
303     std::thread transmitterPortSubset_t(&AUVClass::subsetScatterers, this, \
304         &scatterer_port, \
305         &this->transmitter_port, \
306         auv_pointing_direction_spherical[1].item<float>());
307     std::thread transmitterStarboardSubset_t(&AUVClass::subsetScatterers, this, \
308         &scatterer_starboard, \
309         &this->transmitter_starboard, \
310         - auv_pointing_direction_spherical[1].item<float>());

```

```

308 // joining the subset threads back
309 transmitterFLSSubset_t.join(); transmitterPortSubset_t.join(); transmitterStarboardSubset_t.join();
310
311 // printing the size of these points before subsetting
312 PRINTDOTS
313 std::cout<<"\t AUVClass: scatterer_fls.coordinates.shape (after) = ";
314     fPrintTensorSize(scatterer_fls.coordinates);
315 std::cout<<"\t AUVClass: scatterer_port.coordinates.shape (after) = ";
316     fPrintTensorSize(scatterer_port.coordinates);
317 std::cout<<"\t AUVClass: scatterer_starboard.coordinates.shape (after) = ";
318     fPrintTensorSize(scatterer_starboard.coordinates);
319
320 // multithreading the saving tensors part.
321 std::thread savetensor_t(fSaveSeafloorScatteres, \
322     scatterer, \
323     scatterer_fls, \
324     scatterer_port, \
325     scatterer_starboard);
326
327 // asking ULAs to simulate signal through multithreading
328 std::thread ulafls_signalsim_t(&ULAClass::nfdc_simulateSignal, \
329     &this->ULA_fls, \
330     &scatterer_fls, \
331     &this->transmitter_fls);
332 std::thread ulaport_signalsim_t(&ULAClass::nfdc_simulateSignal, \
333     &this->ULA_port, \
334     &scatterer_port, \
335     &this->transmitter_port);
336 std::thread ulastarboard_signalsim_t(&ULAClass::nfdc_simulateSignal, \
337     &this->ULA_starboard, \
338     &scatterer_starboard, \
339     &this->transmitter_starboard);
340
341 // joining them back
342 ulafls_signalsim_t.join(); // joining back the thread for ULA-FLS
343 ulaport_signalsim_t.join(); // joining back the signals-sim thread for ULA-Port
344 ulastarboard_signalsim_t.join(); // joining back the signal-sim thread for ULA-Starboard
345 savetensor_t.join(); // joining back the signal-sim thread for tensor-saving
346
347 // saving the tensors
348 if (true) {
349     // saving the ground-truth
350     torch::save(this->ULA_fls.signalMatrix,
351         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_fls.pt");
352     torch::save(this->ULA_port.signalMatrix,
353         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_port.pt");
354     torch::save(this->ULA_starboard.signalMatrix,
355         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_starboard.pt");
356 }
357
358 }
359
360 // Imaging Function
361 void image(){
362
363     // asking ULAs to decimate the signals obtained at each time step
364     std::thread ULA_fls_image_t(&ULAClass::nfdc_decimateSignal, \
365         &this->ULA_fls, \
366         &this->transmitter_fls);
367     std::thread ULA_port_image_t(&ULAClass::nfdc_decimateSignal, \
368         &this->ULA_port, \
369         &this->transmitter_port);
370     std::thread ULA_starboard_image_t(&ULAClass::nfdc_decimateSignal, \
371         &this->ULA_starboard, \
372         &this->transmitter_starboard);
373
374     // joining the threads back
375     ULA_fls_image_t.join(); ULA_port_image_t.join(); ULA_starboard_image_t.join();
376
377     // performing the beamforming

```

```

375     std::thread ULA_fls_beamforming_t(&ULAClass::nfdc_beamforming, \
376                                     &this->ULA_fls, \
377                                     &this->transmitter_fls);
378     std::thread ULA_port_beamforming_t(&ULAClass::nfdc_beamforming, \
379                                     &this->ULA_port, \
380                                     &this->transmitter_port);
381     std::thread ULA_starboard_beamforming_t(&ULAClass::nfdc_beamforming, \
382                                             &this->ULA_starboard, \
383                                             &this->transmitter_starboard);
384
385     // joining the filters back
386     ULA_fls_beamforming_t.join(); ULA_port_beamforming_t.join(); ULA_starboard_beamforming_t.join();
387 }
388
389
390
391 /* =====
392 Aim: Init
393 -----*/
394 void init(){
395
396     // call sync-component attributes
397     this->syncComponentAttributes();
398
399     // precomputing delay-matrices for the ULA-class
400     std::thread ULA_fls_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
401                                             &this->ULA_fls, \
402                                             &this->transmitter_fls);
403     std::thread ULA_port_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
404                                             &this->ULA_port, \
405                                             &this->transmitter_port);
406     std::thread ULA_starboard_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
407                                                    &this->ULA_starboard, \
408                                                    &this->transmitter_starboard);
409
410     // joining the threads back
411     ULA_fls_precompute_weights_t.join();
412     ULA_port_precompute_weights_t.join();
413     ULA_starboard_precompute_weights_t.join();
414 }
415
416
417
418 };

```

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  #include <torch/torch.h>
5  #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/ScattererClass.h"
6
7  #ifndef DEVICE
8      // #define DEVICE      torch::kMPS
9      #define DEVICE      torch::kCPU
10 #endif
11
12
13 // adding terrain features
14 #define BOXES      false
15 #define TERRAIN      false
16 #define DEBUG_SEAFLOOR false
17
18
19
20 // Adding boxes
21 void fCreateBoxes(float across_track_length, \
22                  float along_track_length, \
23                  torch::Tensor& box_coordinates, \
24                  torch::Tensor& box_reflectivity){
25
26     // converting arguments to torch tensors
27
28     // setting up parameters
29     float min_width      = 2;      // minimum across-track dimension of the boxes in the sea-floor
30     float max_width      = 5;      // maximum across-track dimension of the boxes in the sea-floor
31
32     float min_length     = 2;      // minimum along-track dimension of the boxes in the sea-floor
33     float max_length     = 5;      // maximum along-track dimension of the boxes in the sea-floor
34
35     float min_height     = 3;      // minimum height of the boxes in the sea-floor
36     float max_height     = 20;     // maximum height of the boxes in the sea-floor
37
38     int meshdensity      = 10;     // number of points per meter.
39     float meshreflectivity = 2;     // average reflectivity of the mesh
40
41     int num_boxes        = 10;     // number of boxes in the sea-floor
42     if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 41\n";
43
44     // finding center point
45     torch::Tensor midxypoints = torch::rand({3, num_boxes}).to(torch::kFloat).to(DEVICE);
46     midxypoints[0]            = midxypoints[0] * across_track_length;
47     midxypoints[1]            = midxypoints[1] * along_track_length;
48     midxypoints[2]            = 0;
49     if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 48\n";
50
51     // assigning dimensions to boxes
52     torch::Tensor boxwidths = torch::rand({num_boxes})*(max_width - min_width) + min_width; // assigning
53     // widths to each boxes
54     torch::Tensor boxlengths = torch::rand({num_boxes})*(max_length - min_length) + min_length; // assigning
55     // lengths to each boxes
56     torch::Tensor boxheights = torch::rand({num_boxes})*(max_height - min_height) + min_height; // assigning
57     // heights to each boxes
58     if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 54\n";
59
60     // creating mesh for each box
61     for(int i = 0; i<num_boxes; ++i){
62
63         // finding x-points
64         torch::Tensor xpoints = torch::linspace(-boxwidths[i].item<float>()/2, \
65         boxwidths[i].item<float>()/2, \

```

```

63         (int)(boxwidths[i].item<float>() * meshdensity));
64     torch::Tensor ypoints = torch::linspace(-boxlengths[i].item<float>()/2, \
65         boxlengths[i].item<float>()/2, \
66         (int)(boxlengths[i].item<float>() * meshdensity));
67     torch::Tensor zpoints = torch::linspace(0, \
68         boxheights[i].item<float>(), \
69         (int)(boxheights[i].item<float>() * meshdensity));
70     if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 69\n";
71
72     // meshgridding
73     auto mesh_grid = torch::meshgrid({xpoints, ypoints, zpoints}, "xy");
74     auto X = mesh_grid[0];
75     auto Y = mesh_grid[1];
76     auto Z = mesh_grid[2];
77     X = torch::reshape(X, {1, X.numel()});
78     Y = torch::reshape(Y, {1, Y.numel()});
79     Z = torch::reshape(Z, {1, Z.numel()});
80     if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 79\n";
81
82     // coordinates
83     torch::Tensor boxcoordinates = torch::cat({X, Y, Z}, 0).to(DEVICE);
84     boxcoordinates[0] = boxcoordinates[0] + midxypoints[0][i];
85     boxcoordinates[1] = boxcoordinates[1] + midxypoints[1][i];
86     boxcoordinates[2] = boxcoordinates[2] + midxypoints[2][i];
87     if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 86\n";
88
89     // creating some reflectivity points too.
90     torch::Tensor boxreflectivity = meshreflectivity + torch::rand({1, boxcoordinates[0].numel()}) - 0.5;
91     boxreflectivity = boxreflectivity.to(DEVICE);
92     if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 90\n";
93
94     // adding to larger matrices
95     if(DEBUG_SEAFLOOR) {std::cout<<"box_coordinates.shape = "; fPrintTensorSize(box_coordinates);}
96     if(DEBUG_SEAFLOOR) {std::cout<<"box_coordinates.shape = "; fPrintTensorSize(boxcoordinates);}
97
98     if(DEBUG_SEAFLOOR) {std::cout<<"box_reflectivity.shape = "; fPrintTensorSize(box_reflectivity);}
99     if(DEBUG_SEAFLOOR) {std::cout<<"boxreflectivity.shape = "; fPrintTensorSize(boxreflectivity);}
100
101     box_coordinates = torch::cat({box_coordinates.to(DEVICE), boxcoordinates}, 1);
102     if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 95\n";
103     box_reflectivity = torch::cat({box_reflectivity.to(DEVICE), boxreflectivity}, 1);
104     if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 97\n";
105 }
106 }
107
108
109
110 // functin that setups the sea-floor
111 void SeafloorSetup(ScattererClass* scatterers) {
112
113     // sea-floor bounds
114     int bed_width = 100; // width of the bed (x-dimension)
115     int bed_length = 100; // length of the bed (y-dimension)
116
117     // multithreading the box creation
118
119     // creating some tensors to pass. This is put outside to maintain scope
120     bool add_boxes_flag = BOXES;
121     torch::Tensor box_coordinates = torch::zeros({3,1}).to(torch::kFloat).to(DEVICE);
122     torch::Tensor box_reflectivity = torch::zeros({1,1}).to(torch::kFloat).to(DEVICE);
123     // std::thread boxes_t(fCreateBoxes, \
124     //     bed_width, bed_length, \
125     //     &box_coordinates, &box_reflectivity);
126     fCreateBoxes(bed_width, \
127         bed_length, \
128         box_coordinates, \
129         box_reflectivity);
130
131     // scatter-intensity
132     // int bed_width_density = 100; // density of points along x-dimension
133     // int bed_length_density = 100; // density of points along y-dimension
134     int bed_width_density = 10; // density of points along x-dimension
135     int bed_length_density = 10; // density of points along y-dimension

```



```

136
137 // setting up coordinates
138 auto xpoints = torch::linspace(0, \
139                               bed_width, \
140                               bed_width * bed_width_density).to(DEVICE);
141 auto ypoints = torch::linspace(0, \
142                               bed_length, \
143                               bed_length * bed_length_density).to(DEVICE);
144
145 // creating mesh
146 auto mesh_grid = torch::meshgrid({xpoints, ypoints}, "ij");
147 auto X          = mesh_grid[0];
148 auto Y          = mesh_grid[1];
149 X              = torch::reshape(X, {1, X.numel()});
150 Y              = torch::reshape(Y, {1, Y.numel()});
151
152 // creating heights of scatterers
153 torch::Tensor Z = torch::zeros({1, Y.numel()}).to(DEVICE);
154
155 // setting up floor coordinates
156 torch::Tensor floorScatter_coordinates = torch::cat({X, Y, Z}, 0);
157 torch::Tensor floorScatter_reflectivity = torch::ones({1, Y.numel()}).to(DEVICE);
158
159 // populating the values of the incoming argument.
160 scatterers->coordinates = floorScatter_coordinates; // assigning coordinates
161 scatterers->reflectivity = floorScatter_reflectivity; // assigning reflectivity
162
163 // // rejoining if multithreading
164 // boxes_t.join(); // bringing thread back
165
166 // combining the values
167 if(DEBUG_SEAFLOOR) std::cout<<"\t SeafloorSetup: line 166 \n";
168 if(DEBUG_SEAFLOOR) {std::cout<<"\t scatterers->coordinates.shape = ";
169                     fPrintTensorSize(scatterers->coordinates);}
170 if(DEBUG_SEAFLOOR) {std::cout<<"\t box_coordinates.shape = "; fPrintTensorSize(box_coordinates);}
171 if(DEBUG_SEAFLOOR) {std::cout<<"\t scatterers->reflectivity.shape = ";
172                     fPrintTensorSize(scatterers->reflectivity);}
173 if(DEBUG_SEAFLOOR) {std::cout<<"\t box_reflectivity = "; fPrintTensorSize(box_reflectivity);}
174
175 // assigning values to the coordinates
176 scatterers->coordinates = torch::cat({scatterers->coordinates, box_coordinates}, 1);
177 scatterers->reflectivity = torch::cat({scatterers->reflectivity, box_reflectivity}, 1);
178
179 }

```

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 = 0.2; // 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
61
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,
17 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,
18 0.1180, 0.1212, 0.1179, 0.1078, 0.0914, 0.0699, 0.0451, 0.0192, -0.0053, -0.0262, -0.0416, -0.0504,
19 -0.0522, -0.0475, -0.0375, -0.0239, -0.0088, 0.0057, 0.0179, 0.0263, 0.0303, 0.0298, 0.0253, 0.0177,
20 0.0086, -0.0008, -0.0091, -0.0153, -0.0187, -0.0191, -0.0168, -0.0123, -0.0065, -0.0004, 0.0052, 0.0095,
21 0.0119, 0.0125, 0.0112, 0.0084, 0.0046, 0.0006, -0.0031, -0.0060, -0.0078, -0.0082, -0.0075, -0.0057,
22 -0.0033, -0.0006, 0.0019, 0.0039, 0.0051, 0.0055, 0.0050, 0.0039, 0.0023, 0.0005, -0.0012, -0.0025,
23 -0.0034, -0.0036, -0.0034, -0.0026, -0.0016, -0.0004, 0.0007, 0.0016, 0.0022, 0.0024, 0.0023, 0.0018,
24 0.0011, 0.0003, -0.0004, -0.0011, -0.0015, -0.0016, -0.0015
25
26
27 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   = 0.1;           // 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,
34     torch::kFloat).to(DEVICE);
35     ULA_direction               = ULA_direction * inter_element_spacing;
36
37     // building the coordinates for the sensors
38     torch::Tensor ULA_coordinates = torch::mul(torch::linspace(0, num_sensors-1, num_sensors).to(DEVICE), \
39     ULA_direction);
40
41     // the coefficients for the decimation filter
42     torch::Tensor lowpassfiltercoefficients =
43     torch::tensor({LOWPASS_DECIMATE_FILTER_COEFFICIENTS}).to(torch::kFloat);
44
45     // assigning values
46     ula_fls->num_sensors      = num_sensors;    // assigning number of sensors
47     ula_fls->inter_element_spacing = inter_element_spacing; // assigning inter-element spacing
48     ula_fls->coordinates      = ULA_coordinates; // assigning ULA coordinates
49     ula_fls->sampling_frequency = sampling_frequency; // assigning sampling frequencys
50     ula_fls->recording_period   = recording_period; // assigning recording period
51     ula_fls->sensorDirection    = ULA_direction; // ULA direction
52     ula_fls->lowpassFilterCoefficientsForDecimation = lowpassfiltercoefficients;
53
54     // assigning values
55     ula_port->num_sensors      = num_sensors;    // assigning number of sensors
56     ula_port->inter_element_spacing = inter_element_spacing; // assigning inter-element spacing
57     ula_port->coordinates      = ULA_coordinates; // assigning ULA coordinates
58     ula_port->sampling_frequency = sampling_frequency; // assigning sampling frequencys
59     ula_port->recording_period   = recording_period; // assigning recording period
60     ula_port->sensorDirection    = ULA_direction; // ULA direction
61     ula_port->lowpassFilterCoefficientsForDecimation = lowpassfiltercoefficients;
62
63     // assigning values
64     ula_starboard->num_sensors      = num_sensors;    // assigning number of sensors
65     ula_starboard->inter_element_spacing = inter_element_spacing; // assigning inter-element spacing
66     ula_starboard->coordinates      = ULA_coordinates; // assigning ULA coordinates
67     ula_starboard->sampling_frequency = sampling_frequency; // assigning sampling frequencys
68     ula_starboard->recording_period   = recording_period; // assigning recording period
69     ula_starboard->sensorDirection    = ULA_direction; // ULA direction
70     ula_starboard->lowpassFilterCoefficientsForDecimation = lowpassfiltercoefficients;
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 }

```

Chapter 9

Reading

9.1 Primary Books

- 1.

9.2 Interesting Papers