Autonomous Underwater Vehicle: A Surveillance Protocol

S.V. Rajendran

February 5, 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 obtaine from that of NOAA ships that are publically 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

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 check-box has not been filled. Since, for the filled ones, the filled scatter will shadow the othersin 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 flag 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.

Beamforming

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

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.

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

#pragma once

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


#define PI            3.14159265


// function to print tensor size
void print_tensor_size(const torch::Tensor& inputTensor) {
    // Printing size
    std::cout << "[";
```

```
    for (const auto& size : inputTensor.sizes()) {
        std::cout << size << ",";
    }
    std::cout << "\b]" <<std::endl;
}

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

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

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

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

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

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

        PRINTSMALLLINE

        // returning os
        return os;
    }

};
```

8.1.2 Class: Transmitter

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

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

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

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

#pragma once

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

#define PI            3.14159265
#define DEBUGMODE_TRANSMITTER    false

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



// control panel
#define ENABLE_RAYTRACING            false








class TransmitterClass{
public:

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

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

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

```

    // shadowing properties
    int azimuthQuantDensity;        // quantization of angles along the azimuth
    int elevationQuantDensity;      // quantization of angles along the elevation
    float rangeQuantSize;           // range-cell size when shadowing
    float azimuthShadowThreshold;   // azimuth thresholding
    float elevationShadowThreshold; // elevation thresholding

    // // shadowing related
    // torch::Tensor checkbox;         // box indicating whether a scatter for a range-angle pair has been found
    // torch::Tensor finalScatterBox;  // a 3D tensor where the third dimension represnets the vector length
    // torch::Tensor finalReflectivityBox; // to store the reflectivity



    // Constructor
    TransmitterClass(torch::Tensor location = torch::zeros({3,1}),
                    torch::Tensor Signal    = torch::zeros({10,1}),
                    float azimuthal_angle   = 0,
                    float elevation_angle   = -30,
                    float azimuthal_beamwidth = 30,
                    float elevation_beamwidth = 30):
                    location(location),
                    Signal(Signal),
                    azimuthal_angle(azimuthal_angle),
                    elevation_angle(elevation_angle),
                    azimuthal_beamwidth(azimuthal_beamwidth),
                    elevation_beamwidth(elevation_beamwidth) {}

    // overloading output
    friend std::ostream& operator<<(std::ostream& os, TransmitterClass& transmitter){
        os<<"\t> azimuth          : "<<transmitter.azimuthal_angle <<std::endl;
        os<<"\t> elevation        : "<<transmitter.elevation_angle <<std::endl;
        os<<"\t> azimuthal beamwidth: "<<transmitter.azimuthal_beamwidth<<std::endl;
        os<<"\t> elevation beamwidth: "<<transmitter.elevation_beamwidth<<std::endl;
        PRINTSMALLLINE
        return os;
    }

    // overloading copyign operator
    TransmitterClass& operator=(const TransmitterClass& other){

        // checking self-assignment
        if(this==&other){
            return *this;
        }

        // allocating memory
        this->location            = other.location;
        this->Signal              = other.Signal;
        this->azimuthal_angle     = other.azimuthal_angle;
        this->elevation_angle     = other.elevation_angle;
        this->azimuthal_beamwidth = other.azimuthal_beamwidth;
        this->elevation_beamwidth = other.elevation_beamwidth;
        this->range               = other.range;

        // transmitted signal attributes
        this->f_low               = other.f_low;
        this->f_high              = other.f_high;
        this->fc                  = other.fc;
        this->bandwidth           = other.bandwidth;

        // shadowing properties
        this->azimuthQuantDensity    = other.azimuthQuantDensity;
        this->elevationQuantDensity  = other.elevationQuantDensity;
        this->rangeQuantSize         = other.rangeQuantSize;
        this->azimuthShadowThreshold = other.azimuthShadowThreshold;
        this->elevationShadowThreshold = other.elevationShadowThreshold;

        // this->checkbox              = other.checkbox;
        // this->finalScatterBox       = other.finalScatterBox;
        // this->finalReflectivityBox  = other.finalReflectivityBox;
```

```

        // returning
        return *this;

    };

    /*==============================================================================
    Aim: Update pointing angle
    --------------------------------------------------------------------------------
    Note:
        > This function updates pointing angle based on AUV's pointing angle
        > for now, we're assuming no roll;
    ------------------------------------------------------------------------------*/
    void updatePointingAngle(torch::Tensor AUV_pointing_vector){

        // calculate yaw and pitch
        if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 140 \n";
        torch::Tensor AUV_pointing_vector_spherical = fCart2Sph(AUV_pointing_vector);
        torch::Tensor yaw                           = AUV_pointing_vector_spherical[0];
        torch::Tensor pitch                         = AUV_pointing_vector_spherical[1];
        if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 144 \n";

        // std::cout<<"\t TransmitterClass: AUV_pointing_vector = "<<torch::transpose(AUV_pointing_vector, 0, 1)<<std::endl;
        // std::cout<<"\t TransmitterClass: AUV_pointing_vector_spherical = "<<torch::transpose(AUV_pointing_vector_spherical, 0, 1)<<std::endl;

        // calculating azimuth and elevation of transmitter object
        torch::Tensor absolute_azimuth_of_transmitter = yaw + torch::tensor({this->azimuthal_angle}).to(torch::kFloat).to(DEVICE);
        torch::Tensor absolute_elevation_of_transmitter = pitch + torch::tensor({this->elevation_angle}).to(torch::kFloat).to(DEVICE);
        if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 149 \n";

        // std::cout<<"\t TransmitterClass: this->azimuthal_angle = "<<this->azimuthal_angle<<std::endl;
        // std::cout<<"\t TransmitterClass: this->elevation_angle = "<<this->elevation_angle<<std::endl;
        // std::cout<<"\t TransmitterClass: absolute_azimuth_of_transmitter = "<<absolute_azimuth_of_transmitter<<std::endl;
        // std::cout<<"\t TransmitterClass: absolute_elevation_of_transmitter = "<<absolute_elevation_of_transmitter<<std::endl;

        // converting back to Cartesian
        torch::Tensor pointing_direction_spherical = torch::zeros({3,1}).to(torch::kFloat).to(DEVICE);
        pointing_direction_spherical[0]         = absolute_azimuth_of_transmitter;
        pointing_direction_spherical[1]         = absolute_elevation_of_transmitter;
        pointing_direction_spherical[2]         = torch::tensor({1}).to(torch::kFloat).to(DEVICE);
        if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 60 \n";

        this->pointing_direction = fSph2Cart(pointing_direction_spherical);
        if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 169 \n";

    }

    /*==============================================================================
    Aim: Subsetting Scatterers inside the cone
    ................................................................................
    steps:
        1. Find azimuth and range of all points.
        2. Fint azimuth and range of current pointing vector.
        3. Subtract azimuth and range of points from that of azimuth and range of current pointing vector
        4. Use tilted ellipse equation to find points in the ellipse
    ------------------------------------------------------------------------------*/
    void subsetScatterers(ScattererClass* scatterers,
                          float tilt_angle){

        // translationally change origin
        scatterers->coordinates = scatterers->coordinates - this->location; if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 188 "<<std::endl;

        /*
        Note: I think something we can do is see if we can subset the matrices by checking coordinate values right away. If one of the coordinate values is x (relative coordiantes), we know for sure that the distance is greater than x, for sure. So, maybe that's something that we can work with
```

```
    */

    // Finding spherical coordinates of scatterers and pointing direction
    torch::Tensor scatterers_spherical      = fCart2Sph(scatterers->coordinates); if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 191 "<<std::endl;
    torch::Tensor pointing_direction_spherical = fCart2Sph(this->pointing_direction); if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 192 "<<std::endl;

    // Calculating relative azimuths and radians
    torch::Tensor relative_spherical = scatterers_spherical - pointing_direction_spherical; if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 199 "<<std::endl;

    // clearing some stuff up
    scatterers_spherical.reset();       if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 202 "<<std::endl;
    pointing_direction_spherical.reset(); if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 203 "<<std::endl;

    // tensor corresponding to switch.
    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;

    // calculating length of axes
    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;
    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;

    // part of calculating the tilted ellipse
    torch::Tensor xcosa   = relative_spherical[0] * torch::cos(tilt_angle_Tensor * PI/180);
    torch::Tensor ysina   = relative_spherical[1] * torch::sin(tilt_angle_Tensor * PI/180);
    torch::Tensor xsina   = relative_spherical[0] * torch::sin(tilt_angle_Tensor * PI/180);
    torch::Tensor ycosa   = relative_spherical[1] * torch::cos(tilt_angle_Tensor * PI/180);
    relative_spherical.reset(); if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 215 "<<std::endl;

    // finding points inside the tilted ellipse
    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;

    // clearing
    xcosa.reset(); ysina.reset(); xsina.reset(); ycosa.reset(); if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 224 "<<std::endl;

    // subsetting points within the elliptical beam
    auto mask                 = (scatter_boolean == 1); // creating a mask
    scatterers->coordinates  = scatterers->coordinates.index({torch::indexing::Slice(), mask});
    scatterers->reflectivity = scatterers->reflectivity.index({torch::indexing::Slice(), mask}); if(DEBUGMODE_TRANSMITTER) std::cout<<"\t\t TransmitterClass: line 229 "<<std::endl;

    // this is where histogram shadowing comes in (later)
    if (ENABLE_RAYTRACING) {rangeHistogramShadowing(scatterers); std::cout<<"\t\t TransmitterClass: line 232 "<<std::endl;}

    // translating back to the points
    scatterers->coordinates = scatterers->coordinates + this->location;

}

/*==========================================================================
Aim: Shadowing method (range-histogram shadowing)
..........................................................................
Note:
    > cut down the number of threads into range-cells
    > for each range cell, calculate histogram
    >
    std::cout<<"\t TransmitterClass: "
--------------------------------------------------------------------------*/
void rangeHistogramShadowing(ScattererClass* scatterers){

    // converting points to spherical coordinates
```

```
    torch::Tensor spherical_coordinates = fCart2Sph(scatterers->coordinates); std::cout<<"\t\t TransmitterClass: line 252 "<<std::endl;

    // finding maximum range
    torch::Tensor maxdistanceofpoints = torch::max(spherical_coordinates[2]); std::cout<<"\t\t TransmitterClass: line 256 "<<std::endl;

    // calculating number of range-cells (verified)
    int numrangecells = std::ceil(maxdistanceofpoints.item<int>()/this->rangeQuantSize);

    // finding range-cell boundaries (verified)
    torch::Tensor rangeBoundaries = \
        torch::linspace(this->rangeQuantSize, \
                        numrangecells * this->rangeQuantSize,\
                        numrangecells); std::cout<<"\t\t TransmitterClass: line 263 "<<std::endl;

    // creating the checkbox (verified)
    int numazimuthcells   = std::ceil(this->azimuthal_beamwidth * this->azimuthQuantDensity);
    int numelevationcells = std::ceil(this->elevation_beamwidth * this->elevationQuantDensity); std::cout<<"\t\t TransmitterClass: line 267 "<<std::endl;

    // finding the deltas
    float delta_azimuth   = this->azimuthal_beamwidth / numazimuthcells;
    float delta_elevation = this->elevation_beamwidth / numelevationcells; std::cout<<"\t\t TransmitterClass: line 271"<<std::endl;

    // creating the centers (verified)
    torch::Tensor azimuth_centers = torch::linspace(delta_azimuth/2, \
                                                    numazimuthcells * delta_azimuth - delta_azimuth/2, \
                                                    numazimuthcells);
    torch::Tensor elevation_centers = torch::linspace(delta_elevation/2, \
                                                    numelevationcells * delta_elevation - delta_elevation/2, \
                                                    numelevationcells); std::cout<<"\t\t TransmitterClass: line 279"<<std::endl;

    // centering (verified)
    azimuth_centers   = azimuth_centers + torch::tensor({this->azimuthal_angle - \
                                                        (this->azimuthal_beamwidth/2)}).to(torch::kFloat);
    elevation_centers = elevation_centers + torch::tensor({this->elevation_angle - \
                                                        (this->elevation_beamwidth/2)}).to(torch::kFloat); std::cout<<"\t\t TransmitterClass: line 285"<<std::endl;

    // building checkboxes
    torch::Tensor checkbox          = torch::zeros({numelevationcells, numazimuthcells}, torch::kBool);
    torch::Tensor finalScatterBox   = torch::zeros({numelevationcells, numazimuthcells, 3}).to(torch::kFloat);
    torch::Tensor finalReflectivityBox = torch::zeros({numelevationcells, numazimuthcells}).to(torch::kFloat); std::cout<<"\t\t TransmitterClass: line 290"<<std::endl;

    // going through each-range-cell
    for(int i = 0; i<(int)rangeBoundaries.numel(); ++i){
        this->internal_subsetCurrentRangeCell(rangeBoundaries[i], \
                                              scatterers,            \
                                              checkbox,              \
                                              finalScatterBox,       \
                                              finalReflectivityBox,  \
                                              azimuth_centers,       \
                                              elevation_centers,     \
                                              spherical_coordinates); std::cout<<"\t\t TransmitterClass: line 301"<<std::endl;

        // after each-range-cell
        torch::Tensor checkboxfilled = torch::sum(checkbox);
        std::cout<<"\t\t\t\t checkbox-filled = "<<checkboxfilled.item<int>()<<"/"<<checkbox.numel()<<" | percent = "<<100 * checkboxfilled.item<float>()/(float)checkbox.numel()<<std::endl;

    }

    // converting from box structure to [3, num-points] structure
    torch::Tensor final_coords_spherical = \
        torch::permute(finalScatterBox, {2, 0, 1}).reshape({3, (int)(finalScatterBox.numel()/3)});
    torch::Tensor final_coords_cart = fSph2Cart(final_coords_spherical); std::cout<<"\t\t
```

```
    TransmitterClass: line 308"<<std::endl;
        std::cout<<"\t\t finalReflectivityBox.shape = "; fPrintTensorSize(finalReflectivityBox);
        torch::Tensor final_reflectivity = finalReflectivityBox.reshape({finalReflectivityBox.numel()}); std::cout<<"\t\t TransmitterClass: line 310"<<std::endl;
        torch::Tensor test_checkbox = checkbox.reshape({checkbox.numel()}); std::cout<<"\t\t TransmitterClass: line 311"<<std::endl;

        // just taking the points corresponding to the filled. Else, there's gonna be a lot of zero zero zero tensors
        auto mask = (test_checkbox == 1); std::cout<<"\t\t TransmitterClass: line 319"<<std::endl;
        final_coords_cart = final_coords_cart.index({torch::indexing::Slice(), mask}); std::cout<<"\t\t TransmitterClass: line 320"<<std::endl;
        final_reflectivity = final_reflectivity.index({mask}); std::cout<<"\t\t TransmitterClass: line 321"<<std::endl;

        // overwriting the scatterers
        scatterers->coordinates  = final_coords_cart;
        scatterers->reflectivity = final_reflectivity; std::cout<<"\t\t TransmitterClass: line 324"<<std::endl;

    }


    void internal_subsetCurrentRangeCell(torch::Tensor rangeupperlimit, \
                                         ScattererClass* scatterers,        \
                                         torch::Tensor& checkbox,           \
                                         torch::Tensor& finalScatterBox,    \
                                         torch::Tensor& finalReflectivityBox, \
                                         torch::Tensor& azimuth_centers,    \
                                         torch::Tensor& elevation_centers,  \
                                         torch::Tensor& spherical_coordinates){

        // finding indices for points in the current range-cell
        torch::Tensor pointsincurrentrangecell = \
            torch::mul((spherical_coordinates[2] <= rangeupperlimit) , \
                       (spherical_coordinates[2] > rangeupperlimit - this->rangeQuantSize));

        // checking out if there are no points in this range-cell
        int num311 = torch::sum(pointsincurrentrangecell).item<int>();
        if(num311 == 0) return;

        // calculating delta values
        float delta_azimuth  = azimuth_centers[1].item<float>() - azimuth_centers[0].item<float>();
        float delta_elevation = elevation_centers[1].item<float>() - elevation_centers[0].item<float>();

        // subsetting points in the current range-cell
        auto mask                                  = (pointsincurrentrangecell == 1); // creating a mask
        torch::Tensor reflectivityincurrentrangecell = scatterers->reflectivity.index({torch::indexing::Slice(), mask});
        pointsincurrentrangecell                   = spherical_coordinates.index({torch::indexing::Slice(), mask});

        // finding number of azimuth sizes and what not
        int numazimuthcells   = azimuth_centers.numel();
        int numelevationcells = elevation_centers.numel();

        // go through all the combinations
        for(int azi_index = 0; azi_index < numazimuthcells; ++azi_index){
            for(int ele_index = 0; ele_index < numelevationcells; ++ele_index){

                // check if this particular azimuth-elevation direction has been taken-care of.
                if (checkbox[ele_index][azi_index].item<bool>()) break;

                // init (verified)
                torch::Tensor current_azimuth = azimuth_centers.index({azi_index});
                torch::Tensor current_elevation = elevation_centers.index({ele_index});

                // // finding azimuth boolean
                // torch::Tensor azi_neighbours = torch::abs(pointsincurrentrangecell[0] - current_azimuth);
                // azi_neighbours              = azi_neighbours <= delta_azimuth; // tinker with this.

                // // finding elevation boolean
                // torch::Tensor ele_neighbours = torch::abs(pointsincurrentrangecell[1] - current_elevation);
                // ele_neighbours              = ele_neighbours <= delta_elevation;
```

```

            // finding azimuth boolean
            torch::Tensor azi_neighbours = torch::abs(pointsincurrentrangecell[0] - current_azimuth);
            azi_neighbours               = azi_neighbours <= this->azimuthShadowThreshold; // tinker with
                this.

            // finding elevation boolean
            torch::Tensor ele_neighbours = torch::abs(pointsincurrentrangecell[1] - current_elevation);
            ele_neighbours               = ele_neighbours <= this->elevationShadowThreshold;


            // combining booleans: means find all points that are within the limits of both the azimuth and
                boolean.
            torch::Tensor neighbours_boolean = torch::mul(azi_neighbours, ele_neighbours);

            // checking if there are any points along this direction
            int num347 = torch::sum(neighbours_boolean).item<int>();
            if (num347 == 0) continue;

            // findings point along this direction
            mask                                = (neighbours_boolean == 1);
            torch::Tensor coords_along_aziele_spherical =
                pointsincurrentrangecell.index({torch::indexing::Slice(), mask});
            torch::Tensor reflectivity_along_aziele =
                reflectivityincurrentrangecell.index({torch::indexing::Slice(), mask});

            // finding the index where the points are at the maximum distance
            int index_where_min_range_is = torch::argmin(coords_along_aziele_spherical[2]).item<int>();
            torch::Tensor closest_coord = coords_along_aziele_spherical.index({torch::indexing::Slice(), \
                                                                index_where_min_range_is});
            torch::Tensor closest_reflectivity = reflectivity_along_aziele.index({torch::indexing::Slice(),
                \
                                                                    index_where_min_range_is});

            // filling the matrices up
            finalScatterBox.index_put_({ele_index, azi_index, torch::indexing::Slice()}, \
                                    closest_coord.reshape({1,1,3}));
            finalReflectivityBox.index_put_({ele_index, azi_index}, \
                                        closest_reflectivity);
            checkbox.index_put_({ele_index, azi_index}, \
                                true);

        }
    }
}




};
```

8.1.3 Class: Uniform Linear Array

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

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


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

#pragma once

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

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

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

// #define DEBUG_ULA true
#define DEBUG_ULA false



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

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

    // decimation-related
    int decimation_factor;
    torch::Tensor lowpassFilterCoefficientsForDecimation; //

    // imaging related
    float range_resolution;           // theoretical range-resolution = $\frac{c}{2B}$
    float azimuthal_resolution;       // theoretical azimuth-resolution =
        $\frac{\lambda}{(N-1)*inter-element-distance}$
    float range_cell_size;            // the range-cell quanta we're choosing for efficiency trade-off
    float azimuth_cell_size;          // the azimuth quanta we're choosing
    torch::Tensor mulFFTMatrix;       // the matrix containing the delays for each-element as a slot
```

```
    torch::Tensor azimuth_centers;   // tensor containing the azimuth centeres
    torch::Tensor range_centers;     // tensor containing the range-centers
    int frame_size;                  // the frame-size corresponding to a range cell in a decimated signal matrix
    torch::Tensor matchFilter;       // torch tensor containing the match-filter

    // constructor
    ULAClass(int numsensors           = 32,
             float inter_element_spacing = 1e-3,
             torch::Tensor coordinates = torch::zeros({3, 2}),
             float sampling_frequency = 48e3,
             float recording_period   = 1,
             torch::Tensor location      = torch::zeros({3,1}),
             torch::Tensor signalMatrix  = torch::zeros({1, 32}),
             torch::Tensor lowpassFilterCoefficientsForDecimation = torch::zeros({1,10})):
             num_sensors(numsensors),
             inter_element_spacing(inter_element_spacing),
             coordinates(coordinates),
             sampling_frequency(sampling_frequency),
             recording_period(recording_period),
             location(location),
             signalMatrix(signalMatrix),
             lowpassFilterCoefficientsForDecimation(lowpassFilterCoefficientsForDecimation){
               // calculating ULA direction
               torch::Tensor sensorDirection = coordinates.slice(1, 0, 1) - coordinates.slice(1, 1, 2);

               // normalizing
               float normvalue = torch::linalg_norm(sensorDirection, 2, 0, true, torch::kFloat).item<float>();
               if (normvalue != 0){
                   sensorDirection = sensorDirection / normvalue;
               }

               // copying direction
               this->sensorDirection = sensorDirection;
        }

    // overrinding printing
    friend std::ostream& operator<<(std::ostream& os, ULAClass& ula){
        os<<"\t number of sensors : "<<ula.num_sensors        <<std::endl;
        os<<"\t inter-element spacing: "<<ula.inter_element_spacing <<std::endl;
        os<<"\t sensor-direction "   <<torch::transpose(ula.sensorDirection, 0, 1)<<std::endl;
        PRINTSMALLLINE
        return os;
    }

    /* ========================================================================
    Aim: Init
    ------------------------------------------------------------------------ */
    void init(TransmitterClass* transmitterObj){

        // calculating range-related parameters
        this->range_resolution    = 1500/(2 * transmitterObj->fc);
        this->range_cell_size     = 40 * this->range_resolution;

        // status printing
        if(DEBUG_ULA) std::cout<<"\t\t ULAClass::init():: this->range_resolution = " << this->range_resolution << std::endl;
        if(DEBUG_ULA) std::cout<<"\t\t ULAClass::init():: this->range_cell_size = " << this->range_cell_size << std::endl;

        // calculating azimuth-related parameters
        this->azimuthal_resolution   = (1500/transmitterObj->fc)/((this->num_sensors-1)*this->inter_element_spacing);
        this->azimuth_cell_size      = 2 * this->azimuthal_resolution;

        // creating and storing the match-filter
        this->nfdc_CreateMatchFilter(transmitterObj);
    }

    // Create match-filter
    void nfdc_CreateMatchFilter(TransmitterClass* transmitterObj){

        // creating matrix for basebanding the signal
```

```
        torch::Tensor basebanding_vector = \
            torch::linspace(0, \
                            transmitterObj->Signal.numel() - 1, \
                            transmitterObj->Signal.numel()).reshape(transmitterObj->Signal.sizes());
        basebanding_vector = \
            torch::exp(COMPLEX_1j * 2 * PI * transmitterObj->fc * basebanding_vector);

        // multiplying the signal with the basebanding vector
        torch::Tensor match_filter = \
            torch::mul(transmitterObj->Signal, basebanding_vector);

        // low-pass filtering
        fConvolve1D(match_filter, this->lowpassFilterCoefficientsForDecimation);

        // creating sampling-indices
        int decimation_factor = std::floor((static_cast<float>(this->sampling_frequency)/2) / \
                                           (static_cast<float>(transmitterObj->bandwidth)/2));
        int final_num_samples =
             std::ceil(static_cast<float>(match_filter.numel())/static_cast<float>(decimation_factor));
        torch::Tensor sampling_indices = \
            torch::linspace(1, \
                            (final_num_samples-1) * decimation_factor,
                            final_num_samples).to(torch::kInt) - torch::tensor({1}).to(torch::kInt);

        // sampling the signal
        match_filter = match_filter.index({sampling_indices});

        // taking conjugate and flipping the signal
        match_filter = torch::flipud( match_filter);
        match_filter = torch::conj(  match_filter);

        // storing the match-filter to the class member
        this->matchFilter = match_filter;
    }

    // overloading the "=" operator
    ULAClass& operator=(const ULAClass& other){
        // checking if copying to the same object
        if(this == &other){
            return *this;
        }

        // copying everything
        this->num_sensors          = other.num_sensors;
        this->inter_element_spacing = other.inter_element_spacing;
        this->coordinates          = other.coordinates.clone();
        this->sampling_frequency   = other.sampling_frequency;
        this->recording_period     = other.recording_period;
        this->sensorDirection      = other.sensorDirection.clone();

        // new additions
        // this->location             = other.location;
        this->lowpassFilterCoefficientsForDecimation = other.lowpassFilterCoefficientsForDecimation;
        // this->sensorDirection      = other.sensorDirection.clone();
        // this->signalMatrix         = other.signalMatrix.clone();


        // returning
        return *this;
    }

    // build sensor-coordinates based on location
    void buildCoordinatesBasedOnLocation(){

        // length-normalize the sensor-direction
        this->sensorDirection = torch::div(this->sensorDirection, torch::linalg_norm(this->sensorDirection, \
                                                                  2, 0, true, \
                                                                  torch::kFloat));
        if(DEBUG_ULA) std::cout<<"\t ULAClass: line 105 \n";

        // multiply with inter-element distance
        this->sensorDirection = this->sensorDirection * this->inter_element_spacing;
        this->sensorDirection = this->sensorDirection.reshape({this->sensorDirection.numel(), 1});
```

```
        if(DEBUG_ULA) std::cout<<"\t ULAClass: line 110 \n";

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


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

    }

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

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

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

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

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

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

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

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

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

        // subtracting to find the relative distances
        torch::Tensor distBetweenScatterersAndSensors = \
            torch::linalg_norm(slottedCoordinates - slottedSensors, 2, 2, true, torch::kFloat);

        // substracting distance between relative fields
```

```
    torch::Tensor distBetweenScatterersAndTransmitter = \
        torch::linalg_norm(slottedCoordinates - slotted_location, 2, 2, true, torch::kFloat);

    // adding up the distances
    torch::Tensor distOfFlight   = distBetweenScatterersAndSensors + distBetweenScatterersAndTransmitter;
    torch::Tensor timeOfFlight   = distOfFlight/1500;
    torch::Tensor samplesOfFlight = torch::floor(timeOfFlight.squeeze() * this->sampling_frequency);

    // Adding pulses
    for(int sensor_index = 0; sensor_index < this->num_sensors; ++sensor_index){
        for(int scatter_index = 0; scatter_index < samplesOfFlight[0].numel(); ++scatter_index){

            // getting the sample where the current scatter's contribution must be placed.
            int where_to_place = samplesOfFlight.index({sensor_index, scatter_index}).item<int>();

            // checking whether that point is out of bounds
            if(where_to_place >= numsamples) continue;

            // placing a point in there
            this->signalMatrix.index_put_({where_to_place, sensor_index}, \
                                  this->signalMatrix.index({where_to_place, sensor_index}) + \
                                  torch::tensor({1}).to(torch::kFloat));

            this->signalMatrix.index_put_({where_to_place, sensor_index}, \
                                  this->signalMatrix.index({where_to_place, sensor_index}) + \
                                   scatterers->reflectivity.index({0, scatter_index}) );
        }
    }

    // Convolving signals with transmitted signal
    torch::Tensor signalTensorAsArgument = \
        transmitterObj->Signal.reshape({transmitterObj->Signal.numel(),1});
    signalTensorAsArgument = torch::tile(signalTensorAsArgument, \
                                  {1, this->signalMatrix.size(1)});

    // convolving the pulse-matrix with the signal matrix
    fConvolveColumns(this->signalMatrix,    \
                  signalTensorAsArgument);

    // trimming the convolved signal since the signal matrix length remains the same
    this->signalMatrix = this->signalMatrix.index({torch::indexing::Slice(0, numsamples), \
                                           torch::indexing::Slice()});

    // printing the shape
    if(DEBUG_ULA) {
        std::cout<<"\t\t\t> this->signalMatrix.shape (after signal sim) = ";
        fPrintTensorSize(this->signalMatrix);
    }

    return;
}

// decimating the obtained signal
void nfdc_decimateSignal(TransmitterClass* transmitterObj){

    // creating the matrix for frequency-shifting
    torch::Tensor integerArray = torch::linspace(0, \
                                           this->signalMatrix.size(0)-1, \
                                           this->signalMatrix.size(0)).reshape({this->signalMatrix.size(0),
                                               1});
    integerArray             = torch::tile(integerArray, {1, this->num_sensors});
    integerArray             = torch::exp(COMPLEX_1j * transmitterObj->fc * integerArray);

    // storing original number of samples
    int original_signalMatrix_numsamples = this->signalMatrix.size(0);

    // printing
    std::cout << "this->signalMatrix.shape = "<< this->signalMatrix.sizes().vec() << std::endl;
    std::cout << "integerArray.shape      = "<< integerArray.sizes().vec()     << std::endl;

    // producing frequency-shifting
    this->signalMatrix       = torch::mul(this->signalMatrix, integerArray);

```

```
    // low-pass filter
    torch::Tensor lowpassfilter_impulseresponse = \
       this->lowpassFilterCoefficientsForDecimation.reshape({this->lowpassFilterCoefficientsForDecimation.numel(),
            1});
    lowpassfilter_impulseresponse = torch::tile(lowpassfilter_impulseresponse, \
                              {1, this->signalMatrix.size(1)});

    // Convolving
    fConvolveColumns(this->signalMatrix, lowpassfilter_impulseresponse);

    // Cutting down the extra-samples from convolution
    this->signalMatrix = \
       this->signalMatrix.index({torch::indexing::Slice(0, original_signalMatrix_numsamples), \
                              torch::indexing::Slice()});

    // building parameters for downsampling
    int decimation_factor        = std::floor(this->sampling_frequency/transmitterObj->bandwidth);
    this->decimation_factor      = decimation_factor;
    int numsamples_after_decimation = std::floor(this->signalMatrix.size(0)/decimation_factor);

    // building the samples which will be subsetted
    torch::Tensor samplingIndices = \
       torch::linspace(0, \
                   numsamples_after_decimation * decimation_factor - 1, \
                   numsamples_after_decimation).to(torch::kInt);

    // downsampling the low-pass filtered signal
    this->signalMatrix = \
       this->signalMatrix.index({samplingIndices, \
                              torch::indexing::Slice()});


}

/* ===========================================================================
Aim: Match-filtering
--------------------------------------------------------------------------- */
void nfdc_matchFilterDecimatedSignal(){
    // Creating a 2D marix out of the signal
    torch::Tensor matchFilter2DMatrix = \
       this->matchFilter.reshape({this->matchFilter.numel(), 1});
    matchFilter2DMatrix = torch::tile(matchFilter2DMatrix, {1, this->num_sensors});

    // 2D convolving to produce the match-filtering
    // std::cout<<"\t\t ULAClass::nfdc_matchFilterDecimatedSignal: this->signalMatrix.shape =
         "<<this->signalMatrix.sizes().vec()<<std::endl;
    fConvolveColumns(this->signalMatrix, matchFilter2DMatrix);

}

/* ===========================================================================
Aim: precompute delay-matrices
--------------------------------------------------------------------------- */
void nfdc_precomputeDelayMatrices(TransmitterClass* transmitterObj){

    // calculating range-related parameters
    int number_of_range_cells    = std::ceil(((this->recording_period * 1500)/2)/this->range_cell_size);
    int number_of_azimuths_to_image = std::ceil(transmitterObj->azimuthal_beamwidth /
         this->azimuth_cell_size);

    // status printing
    if(DEBUG_ULA) std::cout << "\t\t\t ULAClass: number_of_range_cells = " << number_of_range_cells  <<
         std::endl;
    if(DEBUG_ULA) std::cout << "\t\t\t ULAClass: number_of_azimuths_to_image = " <<
         number_of_azimuths_to_image << std::endl;

    // find the centers of the range-cells.
    torch::Tensor range_centers = \
       torch::linspace(this->azimuth_cell_size/2, \
                   (number_of_range_cells - 0.5)*this->azimuth_cell_size, \
                   number_of_range_cells);
    this->range_centers = range_centers;
    if(DEBUG_ULA) std::cout<<"range_centers.sizes().vec() = "<<range_centers.sizes().vec()<<std::endl;
```

```
        // finding the centers of azimuth-cells
        torch::Tensor azimuth_centers = \
            torch::linspace(this->range_cell_size/2, \
                            (number_of_azimuths_to_image - 0.5) * this->range_cell_size, \
                            number_of_azimuths_to_image);
        this->azimuth_centers = azimuth_centers;
        if(DEBUG_ULA) std::cout<<"azimuth_centers.sizes().vec() = "<<azimuth_centers.sizes().vec()<<std::endl;

        // finding the mesh values
        auto range_azimuth_meshgrid = \
            torch::meshgrid({range_centers, azimuth_centers}, "ij");
        torch::Tensor range_grid = range_azimuth_meshgrid[0]; // the columns are range_centers
        torch::Tensor azimuth_grid = range_azimuth_meshgrid[1]; // the rows are azimuth-centers

        // printing
        if(DEBUG_ULA) std::cout << "range_grid.sizes().vec() = " << range_grid.sizes().vec() << std::endl;
        if(DEBUG_ULA) std::cout << "azimuth_grid.sizes().vec() = " << azimuth_grid.sizes().vec() << std::endl;

        // going from 2D to 3D
        range_grid = torch::tile(range_grid.reshape({range_grid.size(0), \
                                                     range_grid.size(1), \
                                                     1}), \
                                 {1,1,this->num_sensors});
        azimuth_grid = torch::tile(azimuth_grid.reshape({azimuth_grid.size(0), \
                                                         azimuth_grid.size(1), \
                                                         1}), \
                                   {1, 1, this->num_sensors});

        // printing
        if(DEBUG_ULA) std::cout << "\t range_grid.sizes().vec() = " << range_grid.sizes().vec() << std::endl;
        if(DEBUG_ULA) std::cout << "\t azimuth_grid.sizes().vec() = " << azimuth_grid.sizes().vec() <<
             std::endl;

        // creating x_m tensor
        torch::Tensor sensorCoordinatesSlot = \
            this->inter_element_spacing * \
            torch::linspace(0, \
                            this->num_sensors - 1,\
                            this->num_sensors).reshape({1,1,this->num_sensors}).to(torch::kFloat);
        sensorCoordinatesSlot = \
            torch::tile(sensorCoordinatesSlot, \
                        {range_grid.size(0),\
                         range_grid.size(1),
                         1});
        if(DEBUG_ULA) std::cout << "\t sensorCoordinatesSlot.sizes().vec() = " <<
             sensorCoordinatesSlot.sizes().vec() << std::endl;

        // calculating distances
        torch::Tensor distanceMatrix =                                   \
            torch::square(range_grid - sensorCoordinatesSlot) +       \
            torch::mul((2 * sensorCoordinatesSlot),                   \
                       torch::mul(range_grid,                         \
                                  1 - torch::cos(azimuth_grid * PI/180)));
        distanceMatrix = \
            torch::sqrt(distanceMatrix);

        // finding the time taken
        torch::Tensor timeMatrix = distanceMatrix/1500;

        // finding the delay to be given
        auto bruh390            = torch::max(timeMatrix, 2, true);
        torch::Tensor max_delay  = std::get<0>(bruh390);
        torch::Tensor delayMatrix = max_delay - timeMatrix;

        // // now that we have the delay entries, we need to create the matrix that does it
        int decimation_factor = \
            std::floor(this->sampling_frequency/transmitterObj->bandwidth);
        this->decimation_factor = decimation_factor;

        // calculating frame-size
        int frame_size = static_cast<float>(std::ceil((2 * this->range_cell_size / 1500)*
             this->sampling_frequency /decimation_factor));
```

```

        // creating the multiplication matrix
        torch::Tensor mulFFTMatrix = \
            torch::linspace(0, \
                            (int)(frame_size)-1, \
                            (int)(frame_size)).reshape({1, \
                                                   (int)(frame_size), \
                                                   1}).to(torch::kFloat);      // creating an array 1,...,frame_size of shape [1,frame_size, 1];
        mulFFTMatrix = \
            torch::div(mulFFTMatrix, \
                       torch::tensor(frame_size).to(torch::kFloat));      // dividing by N
        mulFFTMatrix = \
            mulFFTMatrix * 2 * PI * -1 * COMPLEX_1j;                       // creating tenosr values for -1j * 2pi * k/N
        mulFFTMatrix = \
            torch::tile(mulFFTMatrix, \
                        {number_of_range_cells * number_of_azimuths_to_image, \
                         1, \
                         this->num_sensors});                              // creating the larger tensor for it


        // populating the matrix
        for(int azimuth_index = 0; azimuth_index<number_of_azimuths_to_image; ++azimuth_index){
            for(int range_index = 0; range_index < number_of_range_cells; ++range_index){
                // finding the delays for sensors
                torch::Tensor currentSensorDelays = \
                    delayMatrix.index({range_index, \
                                       azimuth_index, \
                                       torch::indexing::Slice()});
                // reshaping it to the target size
                currentSensorDelays = \
                    currentSensorDelays.reshape({1, \
                                                 1, \
                                                 this->num_sensors});
                // tiling across the plane
                currentSensorDelays = \
                    torch::tile(currentSensorDelays, \
                                {1, frame_size, 1});
                // multiplying across the appropriate plane
                int index_to_place_at = \
                    azimuth_index * number_of_range_cells + \
                    range_index;
                mulFFTMatrix.index_put_({index_to_place_at, \
                                         torch::indexing::Slice(),
                                         torch::indexing::Slice()}, \
                                         currentSensorDelays);
            }
        }

        // std::cout<<"\t\t\t mulFFTMatrix.sizes().vec() = "<<mulFFTMatrix.sizes().vec()<<std::endl;


        // storing the mulFFTMatrix
        this->mulFFTMatrix = mulFFTMatrix;



    }

    // beamforming the signal
    void nfdc_beamforming(TransmitterClass* transmitterObj){

        // ensuring the signal matrix is in the shape we want
        if(this->signalMatrix.size(1) != this->num_sensors)
            throw std::runtime_error("The second dimension doesn't correspond to the number of sensors \n");


        // calculating frame-size from range-cell size
        int frame_size =
            std::ceil((2 * static_cast<float>(this->range_cell_size)/1500) *    \
                        (static_cast<float>(this->sampling_frequency)/static_cast<float>(this->decimation_factor)));
        this->frame_size = frame_size;
```

```
    // adding the batch-dimension
    /* This is to accomodate a particular property of torch library.
    In torch, the first dimension is always the batch-related dimension.
    So in order to use the function torch::bmm(), we need to ensure that the first dimension is that of
        shape. */
    this->signalMatrix = \
        this->signalMatrix.reshape({1, \
                                    this->signalMatrix.size(0), \
                                    this->signalMatrix.size(1)});


    // zero-padding to ensure correctness
    int ideal_length                          = std::ceil(this->range_centers.numel() * frame_size);
    int num_zeros_to_pad_signal_along_dimension_0 = ideal_length - this->signalMatrix.size(1);
    torch::Tensor zero_tensor                 = torch::zeros({this->signalMatrix.size(0),                \
                                                              num_zeros_to_pad_signal_along_dimension_0, \
                                                              this->num_sensors}).to(torch::kFloat);
    this->signalMatrix                        = torch::cat({this->signalMatrix,                          \
                                                            zero_tensor}, 1);


    // breaking the signal into frames
    fBuffer2D(this->signalMatrix, frame_size);


    // tiling it to ensure that it works for all range-angle combinations
    int number_of_azimuths_to_image = this->azimuth_centers.numel();
    this->signalMatrix = \
        torch::tile(this->signalMatrix, \
                    {number_of_azimuths_to_image, 1, 1});


    // element-wise multiplying the signals
    this->signalMatrix = torch::mul(this->signalMatrix, this->mulFFTMatrix);
    this->signalMatrix = torch::sum(this->signalMatrix, 2, true);
    // this->signalMatrix = torch::sum(this->signalMatrix, 2, false);


    // printing
    std::cout<< "this->signalMatrix.sizes().vec() = " << this->signalMatrix.sizes().vec() << std::endl;
    // std::cout<< "this->signalMatrix.sizes().vec() = " << this->signalMatrix.sizes().vec() << std::endl;





    // creating a range-angle mesh for this

    // find the different angles for which we're beamforming.

    // find the delays for the different range-angle combinations

    // splitting the signals into the different range-cells

    // loop for beamforming all of em
}
```

```
};
```

8.1.4 Class: Autonomous Underwater Vehicle

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

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


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

#pragma once

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


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

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

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

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

    // plotting tensors
    fPlotTensors();

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

    //    // getting time ID
```

```
    //    auto timeID = fGetCurrentTimeFormatted();

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

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


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

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

    //    // plotting tensors
    //    fPlotTensors();

    //    // indicating end of thread
    //    std::cout<<"\t\t\t\t\t\t\t Ended (timeID: "<<timeID<<")"<<std::endl;
    // }
}

// including class-definitions
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/ScattererClass.h"

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

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

#define PI           3.14159265
// #define DEBUGMODE_AUV true
#define DEBUGMODE_AUV false


class AUVClass{
public:
    // Intrinsic attributes
    torch::Tensor location;         // location of vessel
    torch::Tensor velocity;         // current speed of the vessel [a vector]
    torch::Tensor acceleration;     // current acceleration of vessel [a vector]
```

```
    torch::Tensor pointing_direction; // direction to which the AUV is pointed

    // uniform linear-arrays
    ULAClass ULA_fls;                // front-looking SONAR ULA
    ULAClass ULA_port;               // mounted ULA [object of class, ULAClass]
    ULAClass ULA_starboard;          // mounted ULA [object of class, ULAClass]

    // transmitters
    TransmitterClass transmitter_fls;   // transmitter for front-looking SONAR
    TransmitterClass transmitter_port;  // mounted transmitter [obj of class, TransmitterClass]
    TransmitterClass transmitter_starboard; // mounted transmitter [obj of class, TransmitterClass]

    // derived or dependent attributes
    torch::Tensor signalMatrix_1;         // matrix containing the signals obtained from ULA_1
    torch::Tensor largeSignalMatrix_1;    // matrix holding signal of synthetic aperture
    torch::Tensor beamformedLargeSignalMatrix;// each column is the beamformed signal at each stop-hop

    // plotting mode
    bool plottingmode;  // to suppress plotting associated with classes

    // spotlight mode related
    torch::Tensor absolute_coords_patch_cart; // cartesian coordinates of patch

    // Synthetic Aperture Related
    torch::Tensor ApertureSensorLocations; // sensor locations of aperture


    /*==========================================================================
    Aim: stepping motion
    ------------------------------------------------------------------------*/
    void step(float timestep){

        // updating location
        this->location = this->location + this->velocity * timestep;
        if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 81 \n";

        // updating attributes of members
        this->syncComponentAttributes();
        if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 85 \n";
    }



    /*==========================================================================
    Aim: updateAttributes
    ------------------------------------------------------------------------*/
    void syncComponentAttributes(){

        // updating ULA attributes
        if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 97 \n";

        // updating locations
        this->ULA_fls.location       = this->location;
        this->ULA_port.location      = this->location;
        this->ULA_starboard.location = this->location;

        // updating the pointing direction of the ULAs
        if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 99 \n";
        torch::Tensor ula_fls_sensor_direction_spherical = fCart2Sph(this->pointing_direction);      // spherical coords
        if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 101 \n";
        ula_fls_sensor_direction_spherical[0]          = ula_fls_sensor_direction_spherical[0] - 90;
        if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 98 \n";
        torch::Tensor ula_fls_sensor_direction_cart    = fSph2Cart(ula_fls_sensor_direction_spherical);
        if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 100 \n";

        this->ULA_fls.sensorDirection       = ula_fls_sensor_direction_cart; // assigning sensor directionf or ULA-FLS
        this->ULA_port.sensorDirection      = -this->pointing_direction;    // assigning sensor direction for ULA-Port
        this->ULA_starboard.sensorDirection = -this->pointing_direction;    // assigning sensor direction for ULA-Starboard
        if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line 105 \n";
```

```cpp

    // // calling the function to update the arguments
    // this->ULA_fls.buildCoordinatesBasedOnLocation();  if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line
        109 \n";
    // this->ULA_port.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: line
        111 \n";
    // this->ULA_starboard.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) std::cout<<"\t AUVClass:
        line 113 \n";

    // updating transmitter locations
    this->transmitter_fls.location    = this->location;
    this->transmitter_port.location   = this->location;
    this->transmitter_starboard.location = this->location;
    if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 102 \n";

    // updating transmitter pointing directions
    this->transmitter_fls.updatePointingAngle(     this->pointing_direction);
    this->transmitter_port.updatePointingAngle(    this->pointing_direction);
    this->transmitter_starboard.updatePointingAngle( this->pointing_direction);
    if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 108 \n";
}



/*===========================================================================
Aim: operator overriding for printing
---------------------------------------------------------------------------*/
friend std::ostream& operator<<(std::ostream& os, AUVClass &auv){
    os<<"\t location = "<<torch::transpose(auv.location, 0, 1)<<std::endl;
    os<<"\t velocity = "<<torch::transpose(auv.velocity, 0, 1)<<std::endl;
    return os;
}


/*===========================================================================
Aim: Subsetting Scatterers
---------------------------------------------------------------------------*/
void subsetScatterers(ScattererClass* scatterers,\
                      TransmitterClass* transmitterObj,\
                      float tilt_angle){

    // ensuring components are synced
    this->syncComponentAttributes();
    if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 120 \n";

    // calling the method associated with the transmitter
    if(DEBUGMODE_AUV) {std::cout<<"\t\t scatterers.shape = "; fPrintTensorSize(scatterers->coordinates);}
    if(DEBUGMODE_AUV) std::cout<<"\t\t tilt_angle = "<<tilt_angle<<std::endl;
    transmitterObj->subsetScatterers(scatterers, tilt_angle);
    if(DEBUGMODE_AUV) std::cout<<"\t AUVClass: page 124 \n";
}

// yaw-correction matrix
torch::Tensor createYawCorrectionMatrix(torch::Tensor pointing_direction_spherical, \
                                        float target_azimuth_deg){

    // building parameters
    torch::Tensor azimuth_correction        =
        torch::tensor({target_azimuth_deg}).to(torch::kFloat).to(DEVICE) - \
                                              pointing_direction_spherical[0];
    torch::Tensor azimuth_correction_radians = azimuth_correction * PI / 180;

    torch::Tensor yawCorrectionMatrix = \
        torch::tensor({torch::cos(azimuth_correction_radians).item<float>(), \
                       torch::cos(torch::tensor({90}).to(torch::kFloat).to(DEVICE)*PI/180 +
                           azimuth_correction_radians).item<float>(), \
                       (float)0,                                          \
                       torch::sin(azimuth_correction_radians).item<float>(), \
                       torch::sin(torch::tensor({90}).to(torch::kFloat).to(DEVICE)*PI/180 +
                           azimuth_correction_radians).item<float>(), \
                       (float)0,                                          \
                       (float)0,                                          \
                       (float)0,                                          \
```

```
                         (float)1}).reshape({3,3}).to(torch::kFloat).to(DEVICE);

        // returning the matrix
        return yawCorrectionMatrix;
    }

    // pitch-correction matrix
    torch::Tensor createPitchCorrectionMatrix(torch::Tensor pointing_direction_spherical, \
                                          float target_elevation_deg){

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

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

        // returning the matrix
        return pitchCorrectionMatrix;
    }

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

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

        // printing size of scatterers before subsetting
        std::cout<< "> AUVClass: Beginning Scatterer Subsetting"<<std::endl;
        std::cout<<"\t AUVClass: scatterer_fls.coordinates.shape (before) = ";
             fPrintTensorSize(scatterer_fls.coordinates);
        std::cout<<"\t AUVClass: scatterer_port.coordinates.shape (before) = ";
             fPrintTensorSize(scatterer_port.coordinates);
        std::cout<<"\t AUVClass: scatterer_starboard.coordinates.shape (before) = ";
             fPrintTensorSize(scatterer_starboard.coordinates);

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

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

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

        // printing the size of these points before subsetting
        PRINTDOTS
```

```
        std::cout<<"\t AUVClass: scatterer_fls.coordinates.shape (after) = ";
            fPrintTensorSize(scatterer_fls.coordinates);
        std::cout<<"\t AUVClass: scatterer_port.coordinates.shape (after) = ";
            fPrintTensorSize(scatterer_port.coordinates);
        std::cout<<"\t AUVClass: scatterer_starboard.coordinates.shape (after) = ";
            fPrintTensorSize(scatterer_starboard.coordinates);

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

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

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

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


    }

    // Imaging Function
    void image(){

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

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

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

```

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


        // // performing the beamforming
        // std::thread ULA_fls_beamforming_t(&ULAClass::nfdc_beamforming,   \
        //                                  &this->ULA_fls,                  \
        //                                  &this->transmitter_fls);
        // std::thread ULA_port_beamforming_t(&ULAClass::nfdc_beamforming,  \
        //                                   &this->ULA_port,                \
        //                                   &this->transmitter_port);
        // std::thread ULA_starboard_beamforming_t(&ULAClass::nfdc_beamforming, \
        //                                        &this->ULA_starboard,      \
        //                                        &this->transmitter_starboard);

        // // joining the filters back
        // ULA_fls_beamforming_t.join();
        // ULA_port_beamforming_t.join();
        // ULA_starboard_beamforming_t.join();

    }


    /* ==========================================================================
    Aim: Init
    ----------------------------------------------------------------------------*/
    void init(){

        // call sync-component attributes
        this->syncComponentAttributes();

        // initializing all the ULAs
        this->ULA_fls.init(      &this->transmitter_fls);
        this->ULA_port.init(     &this->transmitter_port);
        this->ULA_starboard.init( &this->transmitter_starboard);

        // precomputing delay-matrices for the ULA-class
        std::thread ULA_fls_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
                                                &this->ULA_fls,                           \
                                                &this->transmitter_fls);
        std::thread ULA_port_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
                                                 &this->ULA_port,                          \
                                                 &this->transmitter_port);
        std::thread ULA_starboard_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
                                                      &this->ULA_starboard,                \
                                                      &this->transmitter_starboard);

        // joining the threads back
        ULA_fls_precompute_weights_t.join();
        ULA_port_precompute_weights_t.join();
        ULA_starboard_precompute_weights_t.join();

    }


};
```

8.2 Setup Scripts

8.2.1 Seafloor Setup

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

```
/* =====================================
Aim: Setup sea floor
=====================================*/
#include <torch/torch.h>
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/ScattererClass.h"

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


// adding terrrain features
#define BOXES         false
#define TERRAIN       false
#define DEBUG_SEAFLOOR false



// Adding boxes
void fCreateBoxes(float across_track_length, \
                  float along_track_length, \
                  torch::Tensor& box_coordinates,\
                  torch::Tensor& box_reflectivity){

    // converting arguments to torch tensos

    // setting up parameters
    float min_width        = 2;     // minimum across-track dimension of the boxes in the sea-floor
    float max_width        = 5;    // maximum across-track dimension of the boxes in the sea-floor

    float min_length       = 2;     // minimum along-track dimension of the boxes in the sea-floor
    float max_length       = 5;    // maximum along-track dimension of the boxes in the sea-floor

    float min_height       = 3;    // minimum height of the boxes in the sea-floor
    float max_height       = 20;    // maximum height of the boxes in the sea-floor

    int meshdensity        = 10;     // number of points per meter.
    float meshreflectivity = 2;     // average reflectivity of the mesh

    int num_boxes          = 10;    // number of boxes in the sea-floor
    if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 41\n";

    // finding center point
    torch::Tensor midxypoints = torch::rand({3, num_boxes}).to(torch::kFloat).to(DEVICE);
    midxypoints[0]           = midxypoints[0] * across_track_length;
    midxypoints[1]           = midxypoints[1] * along_track_length;
    midxypoints[2]           = 0;
    if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 48\n";

    // assigning dimensions to boxes
    torch::Tensor boxwidths = torch::rand({num_boxes})*(max_width - min_width) + min_width; // assigning widths to each boxes
    torch::Tensor boxlengths = torch::rand({num_boxes})*(max_length - min_length) + min_length; // assigning lengths to each boxes
    torch::Tensor boxheights = torch::rand({num_boxes})*(max_height - min_height) + min_height; // assigning heights to each boxes
    if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 54\n";

    // creating mesh for each box
    for(int i = 0; i<num_boxes; ++i){

        // finding x-points
        torch::Tensor xpoints = torch::linspace(-boxwidths[i].item<float>()/2, \
                                                boxwidths[i].item<float>()/2, \
```

```
                                            (int)(boxwidths[i].item<float>() * meshdensity));
        torch::Tensor ypoints = torch::linspace(-boxlengths[i].item<float>()/2, \
                                            boxlengths[i].item<float>()/2, \
                                            (int)(boxlengths[i].item<float>() * meshdensity));
        torch::Tensor zpoints = torch::linspace(0, \
                                           boxheights[i].item<float>(),\
                                           (int)(boxheights[i].item<float>() * meshdensity));
        if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 69\n";

        // meshgridding
        auto mesh_grid = torch::meshgrid({xpoints, ypoints, zpoints}, "xy");
        auto X         = mesh_grid[0];
        auto Y         = mesh_grid[1];
        auto Z         = mesh_grid[2];
        X              = torch::reshape(X, {1, X.numel()});
        Y              = torch::reshape(Y, {1, Y.numel()});
        Z              = torch::reshape(Z, {1, Z.numel()});
        if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 79\n";

        // coordinates
        torch::Tensor boxcoordinates = torch::cat({X, Y, Z}, 0).to(DEVICE);
        boxcoordinates[0] = boxcoordinates[0] + midxypoints[0][i];
        boxcoordinates[1] = boxcoordinates[1] + midxypoints[1][i];
        boxcoordinates[2] = boxcoordinates[2] + midxypoints[2][i];
        if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 86\n";

        // creating some reflectivity points too.
        torch::Tensor boxreflectivity = meshreflectivity + torch::rand({1, boxcoordinates[0].numel()}) - 0.5;
        boxreflectivity = boxreflectivity.to(DEVICE);
        if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 90\n";

        // adding to larger matrices
        if(DEBUG_SEAFLOOR) {std::cout<<"box_coordinates.shape = "; fPrintTensorSize(box_coordinates);}
        if(DEBUG_SEAFLOOR) {std::cout<<"box_coordinates.shape = "; fPrintTensorSize(boxcoordinates);}

        if(DEBUG_SEAFLOOR) {std::cout<<"box_reflectivity.shape = "; fPrintTensorSize(box_reflectivity);}
        if(DEBUG_SEAFLOOR) {std::cout<<"boxreflectivity.shape = "; fPrintTensorSize(boxreflectivity);}

        box_coordinates   = torch::cat({box_coordinates.to(DEVICE), boxcoordinates}, 1);
        if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 95\n";
        box_reflectivity  = torch::cat({box_reflectivity.to(DEVICE), boxreflectivity}, 1);
        if(DEBUG_SEAFLOOR) std::cout<<"\t fCreateBoxes: line 97\n";
    }
}



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

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

    // multithreading the box creation

    // creating some tensors to pass. This is put outside to maintain scope
    bool add_boxes_flag = BOXES;
    torch::Tensor box_coordinates = torch::zeros({3,1}).to(torch::kFloat).to(DEVICE);
    torch::Tensor box_reflectivity = torch::zeros({1,1}).to(torch::kFloat).to(DEVICE);
    // std::thread boxes_t(fCreateBoxes, \
    //                   bed_width, bed_length, \
    //                   &box_coordinates, &box_reflectivity);
    fCreateBoxes(bed_width, \
                bed_length, \
                box_coordinates, \
                box_reflectivity);

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

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

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

    // creating heights of scattereres
    torch::Tensor Z = torch::zeros({1, Y.numel()}).to(DEVICE);

    // setting up floor coordinates
    torch::Tensor floorScatter_coordinates = torch::cat({X, Y, Z}, 0);
    torch::Tensor floorScatter_reflectivity = torch::ones({1, Y.numel()}).to(DEVICE);

    // populating the values of the incoming argument.
    scatterers->coordinates  = floorScatter_coordinates; // assigning coordinates
    scatterers->reflectivity = floorScatter_reflectivity;// assigning reflectivity

    // // rejoining if multithreading
    // boxes_t.join();// bringing thread back

    // combining the values
    if(DEBUG_SEAFLOOR) std::cout<<"\t SeafloorSetup: line 166 \n";
    if(DEBUG_SEAFLOOR) {std::cout<<"\t scatterers->coordinates.shape = ";
        fPrintTensorSize(scatterers->coordinates);}
    if(DEBUG_SEAFLOOR) {std::cout<<"\t box_coordinates.shape = "; fPrintTensorSize(box_coordinates);}
    if(DEBUG_SEAFLOOR) {std::cout<<"\t scatterers->reflectivity.shape = ";
        fPrintTensorSize(scatterers->reflectivity);}
    if(DEBUG_SEAFLOOR) {std::cout<<"\t box_reflectivity = "; fPrintTensorSize(box_reflectivity);}


    // assigning values to the coordinates
    scatterers->coordinates  = torch::cat({scatterers->coordinates, box_coordinates}, 1);
    scatterers->reflectivity = torch::cat({scatterers->reflectivity, box_reflectivity}, 1);


}
```

8.2.2 Transmitter Setup

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

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

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



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

```

```
    // Setting up transmitter
    float sampling_frequency = 160e3;                // sampling frequency
    float f1                 = 50e3;                 // first frequency of LFM
    float f2                 = 70e3;                 // second frequency of LFM
    float fc                 = (f1 + f2)/2;          // finding center-frequency
    float bandwidth          = std::abs(f2 - f1); // bandwidth
    float pulselength        = 0.2;                  // time of recording

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


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

    float elevation_angle               = -60;               // initial pointing direction

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

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

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

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



    // transmitter-fls
    transmitter_fls->location               = location;           // Assigning location
    transmitter_fls->Signal                 = Signal;             // Assigning signal
    transmitter_fls->azimuthal_angle        = azimuthal_angle_fls; // assigning azimuth angle
    transmitter_fls->elevation_angle        = elevation_angle;     // assigning elevation angle
    transmitter_fls->azimuthal_beamwidth    = azimuthal_beamwidth_fls; // assigning azimuth-beamwidth
    transmitter_fls->elevation_beamwidth    = elevation_beamwidth_fls; // assigning elevation-beamwidth
    // updating quantization densities
    transmitter_fls->azimuthQuantDensity    = azimuthQuantDensity;     // assigning azimuth quant density
    transmitter_fls->elevationQuantDensity  = elevationQuantDensity;   // assigning elevation quant density
    transmitter_fls->rangeQuantSize         = rangeQuantSize;          // assigning range-quantization
    transmitter_fls->azimuthShadowThreshold = azimuthShadowThreshold;  // azimuth-threshold in shadowing
    transmitter_fls->elevationShadowThreshold = elevationShadowThreshold; // elevation-threshold in shadowing
    // signal related
    transmitter_fls->f_low                  = f1;         // assigning lower frequency
    transmitter_fls->f_high                 = f2;         // assigning higher frequency
    transmitter_fls->fc                     = fc;         // assigning center frequency
    transmitter_fls->bandwidth              = bandwidth;  // assigning bandwidth



    // transmitter-portside
    transmitter_port->location              = location;                 // Assigning location
    transmitter_port->Signal                = Signal;                   // Assigning signal
    transmitter_port->azimuthal_angle       = azimuthal_angle_port;     // assigning azimuth angle
    transmitter_port->elevation_angle       = elevation_angle;          // assigning elevation angle
    transmitter_port->azimuthal_beamwidth   = azimuthal_beamwidth_port;  // assigning azimuth-beamwidth
```

```
    transmitter_port->elevation_beamwidth   = elevation_beamwidth_port;  // assigning elevation-beamwidth
    // updating quantization densities
    transmitter_port->azimuthQuantDensity   = azimuthQuantDensity;       // assigning azimuth quant density
    transmitter_port->elevationQuantDensity = elevationQuantDensity;     // assigning elevation quant density
    transmitter_port->rangeQuantSize        = rangeQuantSize;            // assigning range-quantization
    transmitter_port->azimuthShadowThreshold = azimuthShadowThreshold;   // azimuth-threshold in shadowing
    transmitter_port->elevationShadowThreshold = elevationShadowThreshold; // elevation-threshold in shadowing
    // signal related
    transmitter_port->f_low                 = f1;                        // assigning lower frequency
    transmitter_port->f_high                = f2;                        // assigning higher frequency
    transmitter_port->fc                    = fc;                        // assigning center frequency
    transmitter_port->bandwidth             = bandwidth;                 // assigning bandwidth



    // transmitter-starboard
    transmitter_starboard->location                = location;                  // assigning location
    transmitter_starboard->Signal                  = Signal;                    // assigning signal
    transmitter_starboard->azimuthal_angle         = azimuthal_angle_starboard; // assigning azimuthal signal
    transmitter_starboard->elevation_angle         = elevation_angle;
    transmitter_starboard->azimuthal_beamwidth     = azimuthal_beamwidth_starboard;
    transmitter_starboard->elevation_beamwidth     = elevation_beamwidth_starboard;
    // updating quantization densities
    transmitter_starboard->azimuthQuantDensity     = azimuthQuantDensity;
    transmitter_starboard->elevationQuantDensity   = elevationQuantDensity;
    transmitter_starboard->rangeQuantSize          = rangeQuantSize;
    transmitter_starboard->azimuthShadowThreshold  = azimuthShadowThreshold;
    transmitter_starboard->elevationShadowThreshold = elevationShadowThreshold;
    // signal related
    transmitter_starboard->f_low                   = f1;         // assigning lower frequency
    transmitter_starboard->f_high                  = f2;         // assigning higher frequency
    transmitter_starboard->fc                      = fc;         // assigning center frequency
    transmitter_starboard->bandwidth               = bandwidth;  // assigning bandwidth

}
```

8.2.3 Uniform Linear Array

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

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


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


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




void ULASetup(ULAClass* ula_fls,
```

```
            ULAClass* ula_port,
            ULAClass* ula_starboard) {

    // setting up ula
    int num_sensors          = 64;                        // number of sensors
    float sampling_frequency = 160e3;                     // sampling frequency
    float inter_element_spacing = 1500/(2*sampling_frequency); // space between samples
    float recording_period   = 0.25;                         // sampling-period

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

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

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

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

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


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


}
```

8.2.4 AUV Setup

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

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

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

```

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

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

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

8.3 Function Definitions

8.3.1 Cartesian Coordinates to Spherical Coordinates

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

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

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


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

#pragma once

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

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

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

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

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


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

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

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

8.3.2 Spherical Coordinates to Cartesian Coordinates

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

#pragma once

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

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


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



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

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

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

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

8.3.3 Column-Wise Convolution

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

#pragma once

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

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


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


    // printing shape
```

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

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


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

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

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

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

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

}
```

8.3.4 Buffer 2D

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

#pragma once

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

// #define DEBUG_Buffer2D true
#define DEBUG_Buffer2D false


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

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

    // padding with zeros in case it is not a perfect multiple
    if(inputMatrix.size(1)%frame_size != 0){
        // padding with zeros
```

```
        int numberofzeroestoadd = frame_size - (inputMatrix.size(1) % frame_size);
        if(DEBUG_Buffer2D) {
            std::cout  << "\t\t\t fBuffer2D: frame_size = "                << frame_size              << std::endl;
            std::cout  << "\t\t\t fBuffer2D: inputMatrix.sizes().vec() = " << inputMatrix.sizes().vec() << std::endl;
            std::cout  << "\t\t\t fBuffer2D: numberofzeroestoadd = "    << numberofzeroestoadd    << std::endl;
        }

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

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

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

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

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

}
```

8.3.5 fAnglesToTensor

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

  // returning value
  return coordinateTensor;
}
```

8.3.6 fCalculateCosine

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

// function to calculate cosine of two tensors
```

```
torch::Tensor fCalculateCosine(torch::Tensor inputTensor1,
                               torch::Tensor inputTensor2)
{
  // column normalizing the the two signals
  inputTensor1 = fColumnNormalize(inputTensor1);
  inputTensor2 = fColumnNormalize(inputTensor2);

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

  // returning the value
  return cosineBetweenVectors;

}
```

8.4 Main Scripts

8.4.1 Signal Simulation

1.

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

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


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

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

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

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

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

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

```cpp
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/SeafloorSetup/SeafloorSetup.cpp"
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/AUVSetup/AUVSetup.cpp"

// functions
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fPrintTensorSize.cpp"
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fSph2Cart.cpp"
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fCart2Sph.cpp"
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fConvolveColumns.cpp"
// #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fBuffer2D.cpp"
// #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fGetCurrentTimeFormatted.cpp"
// #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fAnglesToTensor.cpp"
// #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fCalculateCosine.cpp"
// #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fColumnNormalize.cpp"
// // #include ""

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

// main-function
int main() {

    // Builing Sea-floor
    ScattererClass SeafloorScatter;
    std::thread scatterThread_t(SeafloorSetup, \
                                &SeafloorScatter);

    // Building ULA
    ULAClass ula_fls, ula_port, ula_starboard;
    std::thread ulaThread_t(ULASetup, \
                            &ula_fls, \
                            &ula_port, \
                            &ula_starboard);

    // Building Transmitter
    TransmitterClass transmitter_fls, transmitter_port, transmitter_starboard;
    std::thread transmitterThread_t(TransmitterSetup,
                                    &transmitter_fls,
                                    &transmitter_port,
                                    &transmitter_starboard);

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

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

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

    // storing
    ScattererClass SeafloorScatter_deepcopy = SeafloorScatter;

    // pre-computing the imaging matrices
    auv.init();

    // mimicking movement
    int number_of_stophops = 1;
    // if (true) return 0;
    for(int i = 0; i<number_of_stophops; ++i){

        // time measuring
        auto start_time = std::chrono::high_resolution_clock::now();

```

```
        // printing some spaces
        PRINTSPACE; PRINTSPACE; PRINTLINE; std::cout<<"i = "<<i<<std::endl; PRINTLINE

        // making the deep copy
        ScattererClass SeafloorScatter  = SeafloorScatter_deepcopy; // copy for FLS

        // simulating the signals received in this time step
        auv.simulateSignal(SeafloorScatter);

        // decimating the signal received in this time step
        auv.image();

        // measuring time
        auto end_time = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> time_duration = end_time - start_time;
        PRINTDOTS; std::cout<<"Time taken (i = "<<i<<") = "<<time_duration.count()<<" seconds"<<std::endl;
             PRINTDOTS

        // moving to next position
        auv.step(0.5);

    }
```

```cpp
// // encapsulating coordinates and reflectivity in a dictionary
// std::unordered_map<std::string, torch::Tensor> floor_scatterers;
// torch::load(floor_scatterers["coordinates"],
//             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/floor_coordinates_3D.pt");
// torch::load(floor_scatterers["reflectivity"],
//             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/floor_scatterers_reflectivity.pt");

// // sending to GPU
// floor_scatterers["coordinates"] = floor_scatterers["coordinates"].to( torch::kMPS);
// floor_scatterers["reflectivity"] = floor_scatterers["reflectivity"].to( torch::kMPS);




// // AUV Setup
// torch::Tensor auv_initial_location       = torch::tensor({0.0, 2.0, 2.0}).view({3,1}).to(torch::kMPS); // initial location
// torch::Tensor auv_initial_velocity       = torch::tensor({1.0, 0.0, 0.0}).view({3,1}).to(torch::kMPS); // initial velocity
// torch::Tensor auv_initial_acceleration   = torch::tensor({0.0, 0.0, 0.0}).view({3,1}).to(torch::kMPS); // initial acceleration
// torch::Tensor auv_initial_pointing_direction = torch::tensor({1.0, 0.0, 0.0}).view({3,1}).to(torch::kMPS); // initial pointing direction

// // Initializing a member of class, AUV
// AUV auv(auv_initial_location,          // assigning initial location
//         auv_initial_velocity,          // assigning initial velocity
//         auv_initial_acceleration,      // assigning initial acceleration
//         auv_initial_pointing_direction); // assigning initial pointing direction



// // Setting up ULAs for the AUV: front, portside and starboard
// const int num_sensors         = 32;               // number of sensors
// const double intersensor_distance = 1e-4;         // distance between sensors

// ULA ula_portside(num_sensors, intersensor_distance); // ULA onbject for portside
// ULA ula_fbls(num_sensors,   intersensor_distance); // ULA object for front-side
// ULA ula_starboard(num_sensors, intersensor_distance); // ULA object for starboard

// auv.ula_portside              = ula_portside;      // attaching portside-ULAs to the AUV
// auv.ula_fbls                  = ula_fbls;          // attaching front-ULA to the AUV
// auv.ula_starboard             = ula_starboard;     // attaching starboard-ULA to the AUV



// // Setting up Projector: front, portside and starboard
// Projector projector_portside(torch::zeros({3,1}).to(torch::kMPS), // location
//                          fDeg2Rad(90),                     // azimuthal angle
//                          fDeg2Rad(-30),                    // elevation angle
//                          fDeg2Rad(30),                     // azimuthal beamwidth
//                          fDeg2Rad(20));                    // elevation beamwidth
// Projector projector_fbls(torch::zeros({3,1}).to(torch::kMPS), // location
//                      fDeg2Rad(0),                          // azimuthal angle
//                      fDeg2Rad(-30),                        // elevation angle
//                      fDeg2Rad(120),                        // azimuthal beamwidth
//                      fDeg2Rad(60));                        // elevation beamwidth;
// Projector projector_starboard(torch::zeros({3,1}).to(torch::kMPS), // location
//                           fDeg2Rad(-90),                   // azimuthal angle
//                           fDeg2Rad(-30),                   // elevation angle
//                           fDeg2Rad(30),                    // azimuthal beamwidth
//                           fDeg2Rad(20));                   // elevation beamwidth;

// auv.projector_portside = projector_portside;        // Attaching projectors to AUV
// auv.projector_fbls    = projector_fbls;             // Attaching projectors to AUV
```

```
  // auv.projector_starboard = projector_starboard;      // Attaching projectors to AUV





  // // testing projection
  // torch::Tensor coordinates = torch::tensor({ 1, 2, 3, 4,
  //                                         0,  0,  0,  0,
  //                                        -1, -1, -1, -1}).view({3,4}).to(torch::kFloat).to(torch::kMPS);
  // torch::Tensor coordinates_normalized = fColumnNormalize(coordinates);
  // torch::Tensor coordinates_projected = coordinates.clone();
  // coordinates_projected[2] = torch::zeros({coordinates.size(1)});

  // torch::Tensor innerproduct = coordinates * coordinates_projected;
  // innerproduct = torch::sum(innerproduct, 0, true);



  // PRINTLINE
  // torch::Tensor xy = coordinates.clone();
  // xy[2] = torch::zeros({xy.size(1)});
  // std::cout<<"coordinates = \n"<<coordinates<<std::endl;
  // PRINTSMALLLINE
  // std::cout<<"xy = \n"<<xy<<std::endl;
  // torch::Tensor xylengths = torch::norm(xy, 2, 0, true, torch::kFloat);
  // std::cout<<"xylengths = \n"<<xylengths<<std::endl;
  // PRINTLINE






  // returning
  return 0;
}
```

Chapter 9

Reading

9.1 Primary Books

1.

9.2 Interesting Papers