# Autonomous Underwater Vehicle: A Surveillance Protocol

S.V. Rajendran

January 28, 2025

# 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.

## 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>

// 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"

#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


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
    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;

    };

    // subsetting scatterers
    void subsetScatterers(ScattererClass* scatterers){


        if (DEBUGMODE_TRANSMITTER) {
            std::cout<<"scatterers->coordinates.shape      = ";
            fPrintTensorSize(scatterers->coordinates);
        }


        // converting from cartesian tensors to spherical tensors
```

```
        torch::Tensor scatterers_spherical = fCart2Sph(scatterers->coordinates);
        if (DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass::subsetScatterers 140 \n";
        scatterers_spherical = scatterers_spherical.to(DEVICE);
        if (DEBUGMODE_TRANSMITTER){
            std::cout<<"scatterers_spherical.shape        = ";
            fPrintTensorSize(scatterers_spherical); PRINTSMALLLINE
        }

        // printing some status
        if (DEBUGMODE_TRANSMITTER){
            PRINTSPACE
            PRINTLINE
            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 > this->azimuthal_beamwidth = " <<this->azimuthal_beamwidth
                 <<std::endl;
            std::cout<<"\t TransmitterClass > this->elevation_beamwidth = " <<this->elevation_beamwidth
                 <<std::endl;
            PRINTLINE
            PRINTSPACE
        }



        if (DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass >reached line 136 \n";

        // finding points that are in the cone
        torch::Tensor scatter_boolean = \
            (torch::square((scatterers_spherical[0] - \
                            torch::tensor({this->azimuthal_angle}).to(torch::kFloat).to(DEVICE)))/torch::square(torch::tensor({this->a
             + \
             torch::square((scatterers_spherical[1] - \
                            torch::tensor({this->elevation_angle}).to(torch::kFloat).to(DEVICE)))/torch::square(torch::tensor({this->e
                                 \
             < 1);

        if (DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass >reached line 141 \n";
        if (DEBUGMODE_TRANSMITTER){
            std::cout<<"scatter_boolean.shape              = ";
            fPrintTensorSize(scatter_boolean);
            PRINTSMALLLINE;
        }

        // subsetting points within the elliptical beam
        auto mask = (scatter_boolean == 1); // creating a mask
        if (DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass >reached line 146 \n";
        scatterers->coordinates = scatterers->coordinates.index({torch::indexing::Slice(), mask});
        if (DEBUGMODE_TRANSMITTER) {
            std::cout<<"scatterers->coordinates.shape             = ";
            fPrintTensorSize(scatterers->coordinates); PRINTSMALLLINE;
        }
        if (DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass >reached line 148 \n";
        scatterers->reflectivity = scatterers->reflectivity.index({torch::indexing::Slice(), mask});
        if (DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass >reached line 150 \n";

        // this is where histogram shadowing comes in (later)


        if (DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass > reached line 156 \n";

    }

};
```

### 8.1.3 Class: Uniform Linear Array

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

```
#include <iostream>
#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


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

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

    // 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):
            num_sensors(numsensors),
            inter_element_spacing(inter_element_spacing),
            coordinates(coordinates),
            sampling_frequency(sampling_frequency),
            recording_period(recording_period) {
               // 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;
       }

    // overriding 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;
    }

    // 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();

        // returning
        return *this;

    }
};
```

### 8.1.4 Class: Autonomous Underwater Vehicle

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

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

#pragma once

// 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 PRINTSPACE   std::cout<<"\n\n\n\n\n\n\n\n"<<std::endl;
// #define PRINTSMALLLINE std::cout<<"------------------------------------------------"<<std::endl;
// #define PRINTLINE    std::cout<<"================================================"<<std::endl;
#define PI           3.14159265
#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;

        // updating attributes of members
        this->updateAttributes();
    }

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

        // updating coordinates of sensors
        this->ULA_fls.coordinates       = this->ULA_fls.coordinates       + this->location;
        this->ULA_port.coordinates      = this->ULA_port.coordinates      + this->location;
        this->ULA_starboard.coordinates = this->ULA_starboard.coordinates + this->location;

        // updating transmitter locations
        this->transmitter_fls     = this->location;
        this->transmitter_port    = this->location;
        this->transmitter_starboard = this->location;
    }



    /*
    ============================================================================
    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: Changing Basis
    Note:
        - The subset-function in the transmitter class assumes the subsetting with the current basis.
        - However, this is not ideal since we want the subsetting to be with respect to the AUV.
        - So this function essentially changes the coordinates of the scatterers to that of the AUV's location and pointing direction.
        - For now, we make the assumption that our AUV doesn't roll. That is, its belly is always to the sea-floor.
        - we apply to the floor-scatterer coordinates, the very operations we need to apply to the pointing vector to make it point in the y-direction.
        - not every operation works though cause the determinant should be one to ensure only rotations and no compressions take place.
        - so we're gonna have to combine three transformations: yaw corrections and pitch correction.
    ----------------------------------------------------------------------------
    */
    void subsetScatterers(ScattererClass* scatterers,\
                          TransmitterClass* transmitterObj){

        // first, translate based on the AUV's current location
        if(DEBUGMODE_AUV) std::cout<<"\t AUV: line 83"<<std::endl;
```

```
        scatterers->coordinates = scatterers->coordinates - this->location;

        // find the azimuth and elevation of pointing-vector
        torch::Tensor pointing_direction_spherical = fCart2Sph(this->pointing_direction);
        pointing_direction_spherical = pointing_direction_spherical.to(DEVICE);
        if(DEBUGMODE_AUV) std::cout<<"\t AUV: line 88"<<std::endl;

        // transforming the matrix accordingly
        torch::Tensor yawCorrectionMatrix = createYawCorrectionMatrix(pointing_direction_spherical, 90);
        if(DEBUGMODE_AUV) std::cout<<"\t AUV: line 144"<<std::endl;
        torch::Tensor pitchCorrectionMatrix = createPitchCorrectionMatrix(pointing_direction_spherical, 0);
        if(DEBUGMODE_AUV) std::cout<<"\t AUV: line 139"<<std::endl;

        // sending both to the right device
        yawCorrectionMatrix   = yawCorrectionMatrix.to(DEVICE);
        pitchCorrectionMatrix = pitchCorrectionMatrix.to(DEVICE);


        // combine the two to minimize MIPS
        torch::Tensor PitchYawCorrectionMatrix = torch::matmul(yawCorrectionMatrix, \
                                                     pitchCorrectionMatrix).to(DEVICE);
        if(DEBUGMODE_AUV) std::cout<<"\t AUV: line 145"<<std::endl;

        // multiply the two with the coordinates to change the coordinates.
        scatterers->coordinates = torch::matmul(PitchYawCorrectionMatrix, \
                                        scatterers->coordinates);
        if(DEBUGMODE_AUV) std::cout<<"\t AUV: line 150"<<std::endl;

        // calling the method associated with the transmitter
        transmitterObj->subsetScatterers(scatterers);
        if(DEBUGMODE_AUV) std::cout<<"\t AUV: line 154"<<std::endl;

        // de-correcting relative rotation-transformations
        yawCorrectionMatrix = createYawCorrectionMatrix(pointing_direction_spherical, \
                                            pointing_direction_spherical[0].item<float>());
        pitchCorrectionMatrix = createPitchCorrectionMatrix(pointing_direction_spherical, \
                                            pointing_direction_spherical[1].item<float>());
        if(DEBUGMODE_AUV) std::cout<<"\t AUV: line 161"<<std::endl;

        // combine the two to minimize MIPS
        PitchYawCorrectionMatrix = torch::matmul(yawCorrectionMatrix, \
                                        pitchCorrectionMatrix).to(DEVICE);
        if(DEBUGMODE_AUV) std::cout<<"\t AUV: line 166"<<std::endl;

        // multiply the two with the coordinates to change the coordinates.
        scatterers->coordinates = torch::matmul(PitchYawCorrectionMatrix, \
                                        scatterers->coordinates);
        if(DEBUGMODE_AUV) std::cout<<"\t AUV: line 171"<<std::endl;

        // de-correcting relative translational transformation
        scatterers->coordinates = scatterers->coordinates + this->location;
        if(DEBUGMODE_AUV) std::cout<<"\t AUV: line 175"<<std::endl;

    }


    // pitch-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;
    }


};
```

## 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


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)

    // scatter-intensity
    int bed_width_density    = 100; // density of points along x-dimension
    int bed_length_density   = 100; // 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({3, Y.numel()}).to(DEVICE);

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

### 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

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

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;
    torch::Tensor Signal  = K * timearray;
    Signal                = torch::mul((f1 + Signal), timearray);
    Signal                = cos(Signal);


    // Setting up transmitter
    torch::Tensor location      = torch::zeros({3,1}).to(DEVICE); // location of transmitter
    float azimuthal_angle_fls    = 90;               // initial pointing direction
    float azimuthal_angle_port   = 180;               // initial pointing direction
    float azimuthal_angle_starboard = 0;              // initial pointing direction

    float elevation_angle       = -60;                // initial pointing direction

    float azimuthal_beamwidth    = 10;                // azimuthal beamwidth of the signal cone
    float elevation_beamwidth    = 10;                // elevation beamwidth of the signal cone

    float azimuthShadowThreshold = 0.5;               // azimuth threshold
    float elevationShadowThreshold = 0.5;             // elevation threshold

    int azimuthQuantDensity   = 20;  // quantization density along azimuth (used for shadowing)
    int elevationQuantDensity = 20;  // quantization density along elevation (used for shadowing)
    float rangeQuantSize      = 20;  // cell-dimension (used for shadowing)



    // populating 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; // assigning azimuth-beamwidth
    transmitter_fls->elevation_beamwidth = elevation_beamwidth; // 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




    // populating 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; // assigning azimuth-beamwidth
    transmitter_port->elevation_beamwidth = elevation_beamwidth; // 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



    // populating transmitter-starboard
    transmitter_starboard->location         = location;
    transmitter_starboard->Signal           = Signal;
    transmitter_starboard->azimuthal_angle  = azimuthal_angle_starboard;
    transmitter_starboard->elevation_angle  = elevation_angle;
    transmitter_starboard->azimuthal_beamwidth = azimuthal_beamwidth;
    transmitter_starboard->elevation_beamwidth = elevation_beamwidth;
    // 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
=====================================*/

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



// =========================================================
void ULASetup(ULAClass* ula_1,
              ULAClass* ula_2) {

    // 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    = 1;                           // sampling-period

    // building the direction for the sensors
    torch::Tensor ULA_direction = torch::tensor({0,1,0}).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);

    // assigning values
    ula_1->num_sensors          = num_sensors;           // assigning number of sensors
    ula_1->inter_element_spacing = inter_element_spacing; // assigning inter-element spacing
    ula_1->coordinates          = ULA_coordinates;       // assigning ULA coordinates
    ula_1->sampling_frequency   = sampling_frequency;    // assigning sampling frequencys
    ula_1->recording_period     = recording_period;      // assigning recording period
    ula_1->sensorDirection      = ULA_direction;         // ULA direction

    ula_1->num_sensors          = num_sensors;           // assigning number of sensors
    ula_1->inter_element_spacing = inter_element_spacing; // assigning inter-element spacing
    ula_1->coordinates          = ULA_coordinates;       // assigning ULA coordinates
    ula_1->sampling_frequency   = sampling_frequency;    // assigning sampling frequencys
    ula_1->recording_period     = recording_period;      // assigning recording period
    ula_1->sensorDirection      = ULA_direction;         // ULA direction

    // assigning values
    ula_2->num_sensors          = num_sensors;           // assigning number of sensors
    ula_2->inter_element_spacing = inter_element_spacing; // assigning inter-element spacing
    ula_2->coordinates          = ULA_coordinates;       // assigning ULA coordinates
    ula_2->sampling_frequency   = sampling_frequency;    // assigning sampling frequencys
    ula_2->recording_period     = recording_period;      // assigning recording period
    ula_2->sensorDirection      = ULA_direction;         // ULA direction

    ula_2->num_sensors          = num_sensors;           // assigning number of sensors
    ula_2->inter_element_spacing = inter_element_spacing; // assigning inter-element spacing
    ula_2->coordinates          = ULA_coordinates;       // assigning ULA coordinates
    ula_2->sampling_frequency   = sampling_frequency;    // assigning sampling frequencys
    ula_2->recording_period     = recording_period;      // assigning recording period
    ula_2->sensorDirection      = ULA_direction;         // ULA direction

}
```

### 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({0,0,30}).reshape({3,1}).to(torch::kFloat).to(DEVICE);; // starting location of AUV
    torch::Tensor velocity           = torch::tensor({1,0, 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

# Chapter 9

# Reading

## 9.1 Primary Books

1.

## 9.2 Interesting Papers