# Autonomous Underwater Vehicle:
# A Surveillance Protocol

S.V. Rajendran

September 27, 2025

# Preface

This project is an attempt at combining all of my major skills into creating a simulation, imaging, perception and control pipeline for Autonomous Underwater Vehicles (AUV). As such, creating this project involves creating a number of pipelines.

The first pipeline is the signal simulation pipeline. The signal simulation pipeline involves sea-floor point-cloud creation and simulating the signals received by the sensor arrays of the AUV. The signals recorded by the sensor-arrays on the AUV contains information from the surrounding environment. The imaging pipeline performs certain operations on the recorded signals to obtain acoustic images of the surrounding environment. To that end, this pipeline involves the topics of signal processing, linear algebra, signals and systems.

As such, the second pipeline is the imaging pipeline. The inputs to the imaging pipeline is the signals recorded by the different sensor-arrays of the AUV, in addition to the parameters of the AUV and its components. This pipeline involves match-filtering, focussing and beamforming operations to create acoustic images of the surrounding environment. Depending on the number of ULAs present, the imaging pipeline is responsible for creating multiple acoustic images in real-time. Thus, this pipeline involves the topics of Digital Signal Processing, Match-Filtering, Estimation and Detection Theory and so on.

The images created by the imaging pipeline are fed to the perception-to-control pipeline. This pipeline takes in the image formed created from the ULA signals, parameters of AUV and its components, and some historical data, it provides instructions regarding the movement of the AUV. The mapping from the inputs to the controls is called policy. Learning policies is a core part of reinforcement learning. Thus, this pipeline mainly involves the topics of reinforcement learning. And since we'll be using convolutional neural nets and transformers for learning the policies, this pipeline involves a significant amount of machine and deep learning.

The final result is an AUV that is primarily trained to map an area of the sea-floor in a constant surveillance mode. The RL-trained policy will also be trained to deal with different kinds of sea-floor terrains: those containing hills, valleys, and path-obstructing features. Due to the resource constrained nature of the marine vessel, we also prioritize efficient policies in the policy-training pipeline.

The project is currently written in C++. And since there is non-trivial amount of training and adaptive features in the pipelines, we'll be using LibTorch (the C++ API of PyTorch) to enable computation graphs, backpropagation and thereby, learning in our AUV pipeline. However, for the sections where a computation graph is not required, such as signal simulation, we will be writing templated STL code.

# 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.
- Or if you do not wish to follow a source-control approach, just download the repository as a zip file after clicking the blue code button.

# Chapter 2

# Underwater Environment Setup

## Overview

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

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

## 2.1 Sea "Floor"

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

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

**Algorithm 1** Hill Creation

**Input:** Mean vector $\mathbf{m}$, Dimension vector $\mathbf{d}$, 2D points $\mathbf{P}$
**Output:** Updated $\mathbf{P}$ with hill heights
num_hills $\leftarrow$ numel($\mathbf{m}_x$)
$H \leftarrow$ Zeros tensor of size $(1, \text{numel}(\mathbf{P}_x))$
**for** $i = 1$ to num_hills **do**
  $x_{\text{norm}} \leftarrow \frac{\frac{\pi}{2}(\mathbf{P}x - \mathbf{m}x[i])}{\mathbf{d}x[i]}$
  $y_{\text{norm}} \leftarrow \frac{\frac{\pi}{2}(\mathbf{P}y - \mathbf{m}y[i])}{\mathbf{d}y[i]}$
  $h_x \leftarrow \cos(x_{\text{norm}}) \cdot e^{\frac{|x\text{norm}|}{10}}$
  $h_y \leftarrow \cos(y_{\text{norm}}) \cdot e^{\frac{|y\text{norm}|}{10}}$
  $h \leftarrow \mathbf{d}_z[i] \cdot h_x \cdot h_y$
  **Apply boundary conditions:**
  **if** $x_{\text{norm}} > \frac{\pi}{2}$ or $x_{\text{norm}} < -\frac{\pi}{2}$ or $y_{\text{norm}} > \frac{\pi}{2}$ or $y_{\text{norm}} < -\frac{\pi}{2}$ **then**
    $h \leftarrow 0$
  **end if**
  $H \leftarrow H + h$
**end for**
$\mathbf{P} \leftarrow$ concatenate$([\mathbf{P}, H])$

## 2.2 Simple Structures

### 2.2.1 Boxes

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

**Algorithm 2** Generate Box Meshes on Sea Floor

**Require:** $across\_track\_length$, $along\_track\_length$, $box\_coordinates$, $box\_reflectivity$

**Initialize** min/max width, length, height, meshdensity, reflectivity, and number of boxes
Generate random center points for boxes:
$midxypoints \leftarrow \text{rand}([3, num\_boxes])$
$midxypoints[0] \leftarrow midxypoints[0] \times across\_track\_length$
$midxypoints[1] \leftarrow midxypoints[1] \times along\_track\_length$
$midxypoints[2] \leftarrow 0$
Assign random dimensions to each box:
$boxwidths \leftarrow \text{rand}(num\_boxes) \times (max\_width - min\_width) + min\_width$
$boxlengths \leftarrow \text{rand}(num\_boxes) \times (max\_length - min\_length) + min\_length$
$boxheights \leftarrow \text{rand}(num\_boxes) \times (max\_height - min\_height) + min\_height$
**for** $i = 1$ to $num\_boxes$ **do**
  Generate mesh points along each axis:
  $xpoints \leftarrow \text{linspace}(-boxwidths[i]/2, boxwidths[i]/2, boxwidths[i] \times meshdensity)$
  $ypoints \leftarrow \text{linspace}(-boxlengths[i]/2, boxlengths[i]/2, boxlengths[i] \times meshdensity)$
  $zpoints \leftarrow \text{linspace}(0, boxheights[i], boxheights[i] \times meshdensity)$
  Generate 3D mesh grid:
  $X, Y, Z \leftarrow \text{meshgrid}(xpoints, ypoints, zpoints)$
  Reshape $X, Y, Z$ into 1D tensors
  Compute final coordinates:
  $boxcoordinates \leftarrow \text{cat}(X, Y, Z)$
  $boxcoordinates[0] \leftarrow boxcoordinates[0] + midxypoints[0][i]$
  $boxcoordinates[1] \leftarrow boxcoordinates[1] + midxypoints[1][i]$
  $boxcoordinates[2] \leftarrow boxcoordinates[2] + midxypoints[2][i]$
  Generate reflectivity values:
  $boxreflectivity \leftarrow meshreflectivity + \text{rand}(1, \text{size}(boxcoordinates)) - 0.5$
  Append data to final tensors:
  $box\_coordinates \leftarrow \text{cat}(box\_coordinates, boxcoordinates, 1)$
  $box\_reflectivity \leftarrow \text{cat}(box\_reflectivity, boxreflectivity, 1)$
**end for**

### 2.2.2 Sphere

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

**Algorithm 3** Sphere Creation

**num_hills** $\leftarrow$ Number of Hills

# Chapter 3

# Hardware Setup

## Overview

The AUV contains a number of hardware that enables its functioning. A real AUV contains enough components to make a victorian child faint. And simulating the whole thing and building pipelines to model their working is not the kind of project to be handled by a single engineer. So we'll only model and simulate those components that are absolutely required for the running of these pipelines.

## 3.1 Transmitter

Probing systems are those systems that send out a signal, listen to the reflection and infer qualitative and quantitative qualities of the environment, matter or object, it was trying to infer information about. The transmitter is one of the most fundamental components of probing systems. As the name suggests, the transmitter is the equipment responsible for sending out the probing signal into the medium.

Transmitters are of many kinds. But the ones that we will be considering will be directed transmitters, which means that these transmitters have an associated beampattern. To the uninitiated, this means that the power of the transmitted signal is not transmitted in all directions equally. A beampattern is a graphical representation of the power received by an ideal receiver when placed at different angles.

Transmitters made out of a linear-array of individual transmitters use beamforming to "direct" the major power of the transmitter. These kind of systems have well studied beampatterns which can be utilized in our simulations. These kind of studies and inculcating that in our pipelines produce accurate signal simulation pipelines.

For now, we stick to a very simple model of a transmitter. We assume that the transmitter sends out the power equally into a particular cone from the AUV position.

The full-script for the setup of the transmitter is given in section **??** and the class definition for the transmitter is given in section **??**.

## 3.2 Uniform Linear Array

Perhaps the most important component of probing systems are the “listening” systems. After “illuminating” the medium with the signal, we need to listen to the reflections in order to infer properties. In fact, there are some probing systems that do not use transmitter. Thus, this easily makes the case for the simple fact that the “listening” components of probing systems are the most important components of the whole system.

Uniform arrays are of many kinds but the most popular ones are uniform linear arrays and uniform planar arrays. The arrays in this case contain a number of sensors arranged in a uniform manner across a line or a plane.

Linear arrays have the property that the information obtained from elevation, $\phi$ is no longer available due to the dimensionality of the array-structure. Thus, the images obtained from processing the signals recorded by a uniform linear array will only have two-dimensions: the azimuth, $\theta$ and the range, $r$.

Thus, for 3D imaging, we shall be working with planar arrays. However, due to the higher dimensionality of the output signal, the class of algorithms required to create 3D images are a lot more computationally efficient. In addition, due to the simpler nature of the protocols involved with our AUV, uniform linear arrays will work just fine.

## 3.3 Marine Vessel

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

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

# Chapter 4

# Signal Simulation

## Overview

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

## 4.1 Transmitted Signal

- In probing systems, which are systems which transmit a signal and infer qualitative and quantiative characterisitics of the environment from the signal return, the ideal signal is the dirac delta signal. However, dirac-deltas are nearly impossible to create because of their infinite bandwidth structure. Thus, we need to use something else that is more practical but at the same time, gets us quite close the diract-delta. So we use something of a watered-down delta-function, which is a bandlimited delta function, or the linear frequency-modulated signal. The LFM is a asignal whose frequency increases linearly in its duration. This means that the signal has a flat magnitude spectrum but quadratic phase.
- The LFM is characterised by the bandwidth and the center-frequency. The higher the resolution required, the higher the transmitted bandwidth is. So bandwidth is a characterizing factor. The higher the bandwidth, the better the resolution obtained.
- The transmitted signals used in these cases depend highly on the kind of SONAR we're using it for. The systems we're using currently contain one FLS and two side-scan or 3 FLS (I'm yet to make up mind here).
- The signal is defined in setup-script of the transmitter. Please refer to section: **??**;

## 4.2 Signal Simulation

1. The signals simulation is performed using simple ray-tracing. The distance travelled from the transmitted to scatterer and then the sensor is calculated for each scatter-sensor pair. And the transmitted signal is placed at the recording of each sensor corresponding to each scatterer.
2. First we obtain the set of scatterers that reflect the transmitted signal.
3. The distance between all the sensors and the scatterer distances are calculated.
4. The time of flight from the transmitter to each scatterer and each sensor is calculated.
5. This time is then calculated into sample number by multiplying with the sampling-frequency of the uniform linear arrays.
6. We then build a signal matrix that has the dimensions corresponding to the number of samples that are recorded and the number of sensors that are present in the sensor-array.
7. We place impulses in the points corresponding to when the signals arrives from the scatterers. The result is a matrix that has x-dimension as the number of samples and the y-dimension as the number of sensors.
8. Each column is then convolved (linearly convolved) with the transmitted signal. The resulting matrix gives us the signal received by each sensor. Note that this method doesn't consider doppler effects. This will be added later.

**Algorithm 4** Signal Simulation

**ScatterCoordinates** $\leftarrow$
**ScatterReflectivity** $\leftarrow$
**AngleDensity** $\leftarrow$ Quantization of angles per degree.
**AzimuthalBeamwidth** $\leftarrow$ Azimuthal Beamwidth
**RangeCellWidth** $\leftarrow$ The range-cell width

## 4.3 Ray Tracing

- There are multiple ways for ray-tracing.
- The method implemented during the FBLS and SS SONARs weren't super efficient as it involved pair-wise dot-products. Which becomes an issue when the number of points are increased, which is the case when the range is super high or the beamwidth is super high.

### 4.3.1 Pairwise Dot-Product

- In this method, given the coordinates of all points that are currently in the illumination cone, we find the cosines between every possible pairs of points.
- This is where the computational complexity arises as the number of dot products increase exponentially with increasing number of points.

- This method is a liability when it comes to situations where the range is super high or when the angle-beamwidth is non-narrow.

### 4.3.2 Range Histogram Method

- Given the angular beamwidths: azimuthal beamwidth and elevation beamwidth, we quantize square cone into a number of different values (note that the square cone is not an issue as the step before ensures conical subsetting).
- We split the points into different ”range-cells”.
- For each range-cell, we make a 2D histogram of azimuths and elevations. Then within each range-cell and for each azimuth-elevation pair, we find the closest point and add it to the check-box.
- In the next range-cell, we only work with those azimuth-elevation pairs whose check-box has not been filled. Since, for the filled ones, the filled scatter will shadow the othersin the following range cells.

---

**Algorithm 5** Range Histogram Method

---

**ScatterCoordinates** $\leftarrow$
**ScatterReflectivity** $\leftarrow$
**AngleDensity** $\leftarrow$ Quantization of angles per degree.
**AzimuthalBeamwidth** $\leftarrow$ Azimuthal Beamwidth
**RangeCellWidth** $\leftarrow$ The range-cell width

---

# Chapter 5

# Imaging

## Overview

- Present basebanding, low-pass filtering and decimation.
- Present beamforming.
- Present different synthetic-aperture concepts.

## 5.1 Decimation

1. Due to the large sampling-frequencies employed in imaging SONAR, it is quite often the case that the amount of samples received for just a couple of milliseconds make for non-trivial data-size.
2. In such cases, we use some smart signal processing to reduce the data-size without loss of information. This is done using the fact that the transmitted signal is non-baseband. THis means that using a method known as quadrature modulation, we can maintain the information content without the humongous amount data.
3. After basebanding the signal, this process involves decimation of the signal respecting the bandwidth of the transmitted signal.

### 5.1.1 Basebanding

1. Basebanding is performed utilizing the frequency-shifting property of the fourier transform

$$x(t)e^{j2\pi\omega_0 t} \leftrightarrow X(\omega - \omega_0)$$

2. Since we're working with digital signals, this is implemented in the following manner

$$x[n]e^{j\frac{2\pi k_0 n}{N}} \leftrightarrow X(k - k_0)$$

**Algorithm 6** Basebanding

**ScatterCoordinates** $\leftarrow$
**ScatterReflectivity** $\leftarrow$
**AngleDensity** $\leftarrow$ Quantization of angles per degree.
**AzimuthalBeamwidth** $\leftarrow$ Azimuthal Beamwidth
**RangeCellWidth** $\leftarrow$ The range-cell width

### 5.1.2 Lowpass filtering

1. Now that we have the signal in the baseband, we lowpass filter the signal based on the bandwidth of the signal. Since we're perfectly centering the signal using $f_c$, we can have the cutoff-frequency of the lowpass filter to be just above half the bandwidth of the transmitted signal. Note that the signals should not be brought down back into the real-domain using abs() or real() functions since the negative frequencies are no longer symmetrical.

2. After low-pass filtering, we have a band-restricted signal that contains all of the data in the baseband. This allows for decimation, which is what we'll do in the next step.

### 5.1.3 Decimation

1. Now that we have the bandlimited signal, what we shall do is decimation. Decimation essentially involves just taking every n-th sample where $n$ in this case is the decimation factor.

2. The resulting signal contains the same information as that of the real-sampled signal but with much less number of samples.

## 5.2 Match-Filtering

1. To understand why match-filtering is going on, it is important to understand pulse compression.

2. In "probing" systems, which are basically systems where we send out some signal, listen to the reflection and infer quantitative and qualitative aspects of the environment, the best signal is the impulse signal (see Dirac Delta). However. this signal is not practical to use. Primarily due to the very simple fact that this particular signal has a flat and infinite bandwidth. However, this signal is the idea.

3. So instead, we're left with using signals that have a finite length, $T_{\text{Transmitted Signal}}$. However, the issue with that is that a scatter of initesimal dimension produce a response that has a length of $T_{\text{Transmitted Signal}}$. Thus, it is important to ensure that the response of each object, scatter or what not has comparable dimensions. This is where pulse compression comes in. Using this technique, we transform the received signal to produce a signal that is as close as possible to the signal we'd receive if we were to send out a diract delta pulse.

4. Thus, this process involves something of a detection. The closest method is something of a correlation filter where we run a copy of the transmitted signal through the received recording and take inner-products at each time step (known as the cor-

relation operation). This method works great if we're in the real domain. However, thanks to the quadrature demodulation we performed, this process is now no longer valid. But the idea remainds the same. The point of doing a correlation analysis is so that where there is a signal, a spike appears. The sample principle is used to develop the match-filter.

5. We want to produce a filter, which when convolved with the received signal produces a spike. Since we're trying to produce something similar to the response of an ideal transmission system, we want the output to be that of an ideal spike, which is the delta function. So we're essentially trying to find a filter, which when multiplied with the transmitted signal, produces the diract delta.

6. The answer can be found by analyzing the frequency domain. The frequency domain basis representation of the delta-function is a flat magnitude and linear phase. Thus, this means that the filter that we use on the transmitted signal must produce a flat magnitude and linear phase. The transmitted signal that we're working with, being an LFM, means that the magnitude is already flat. The phase, however, is quadratic. So we need the matched filter to have a flat magnitude and a quadratic phase that cancels away that of the transmitted signa's quadratic component. All this leads to the best candidate: the complex conjugate of the transmitted signal. However, since we're now working with the quadrature demodulated signal, the matched filter is the complex conjugate of the quadrature demodulated transmitted signal.

7. So once the filter is made, convolving that with the received signal produces a number of spikes in the processed signal. Note that due to working in the digital domain and some other factors, the spikes will not be perfect. Thus it is not safe to take the abs() or real() just yet. We'll do that after beamforming.

8. But so far, this marks the first step of the perception pipeline.

**Algorithm 7** Match-Filtering

**ScatterCoordinates** $\leftarrow$
**ScatterReflectivity** $\leftarrow$
**AngleDensity** $\leftarrow$ Quantization of angles per degree.
**AzimuthalBeamwidth** $\leftarrow$ Azimuthal Beamwidth
**RangeCellWidth** $\leftarrow$ The range-cell width

## Beamforming

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

**Algorithm 8** Beamforming

**ScatterCoordinates** $\leftarrow$
**ScatterReflectivity** $\leftarrow$
**AngleDensity** $\leftarrow$ Quantization of angles per degree.
**AzimuthalBeamwidth** $\leftarrow$ Azimuthal Beamwidth
**RangeCellWidth** $\leftarrow$ The range-cell width

# Chapter 6

# Control Pipeline

## Overview

1. The inputs to the control-pipeline is the images obtained from previous pipeline.
2. Currently the plan is to use DQN.

## DQN

1. Here we're essentially trying to create a control pipeline that performs the protocol that we need.
2. The aim of the AUV is to continuously map a particular area of the sea-floor and perform it despite the presence of sea-floor structures.
3. 

**Algorithm 9** DQN

**ScatterCoordinates** $\leftarrow$
**ScatterReflectivity** $\leftarrow$
**AngleDensity** $\leftarrow$ Quantization of angles per degree.
**AzimuthalBeamwidth** $\leftarrow$ Azimuthal Beamwidth
**RangeCellWidth** $\leftarrow$ The range-cell width

### Artificial Acoustic Imaging

1. In order to ensure faster development, we shall start off with training the DQN algorithm with artificial acoustic images. This is rather important due to the fact that the imaging pipelines (currently) has some non-trivial latency. This means that using those pipelines to create the inputs to the DQN algorithm will skyrocket the training time.
2. So the approach that we shall be taking will be write functions to create artifical acoustic images directly from the scatterer-coordinates and scatterer-reflectivity values. The latency for these functions are negligible compared to that of beamforming-

based imaging algorithms. The function for this has been added and is available in section **??** under the function name, *nfdc_createAcousticImage*. Please note that these functions are not to be directly called from the main function. Instead, it is expected that the main function calls the AUV classes's method, *create_ArtificialAcousticImage*. This function calls the class ULA's method appropriately.

3. After the ULA's create their respective acoustic images, they are put together, either by dimension-wise concatenation or depth-wise concatenation and feed to the neural net to produce control sequences.

4. We need to work on the dimensions of these images though. The best thing to do right now is to finalize the transmitter and receiver parameters and then over-estimate the dimensions of the final beamforming-produced image. We shall then use these dimensions to create the artificial acoustic image and start training the policy.

**Algorithm 10** Artifical Acoustic Imaging

**ScatterCoordinates** ← Coordinates of points in the point-cloud.
**auvCoordinates** ← Coordinates of AUV/ULA.

# Chapter 7

# Results

# Chapter 8

# Software

## Overview

- 

## 8.1 Class Definitions

### 8.1.1 Class: Scatter

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

```
// // 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);

//         // returning os
//         return os;
//     }

//     // copy constructor from a pointer
//     ScattererClass(ScattererClass* scatterers){

//         // copying the values
//         this->coordinates = scatterers->coordinates;
//         this->reflectivity = scatterers->reflectivity;
//     }

// };

template <typename T>
class ScattererClass
{
public:
    // members
    std::vector<std::vector<T>> coordinates;
    std::vector<T>            reflectivity;

    // Constructor
    ScattererClass()  {}

    // Constructor
    ScattererClass(std::vector<std::vector<T>> coordinates_arg,
                   std::vector<T>            reflectivity_arg):
                   coordinates(coordinates_arg),
                   reflectivity(reflectivity_arg) {}

    // Save to CSV
    void savetocsv(){
        fWriteMatrix(this->coordinates, "../csv-files/coordinates.csv");
        fWriteVector(this->reflectivity, "../csv-files/reflectivity.csv");
    }

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

    //     // printing coordinate shape
    //     os << format("\t> scatterer.coordinates.shape = [{}, {}]\n", scatterer.coordinates.size(),
         scatterer.coordinates[0].size());

```

```
    //    // printing reflectivity shape
    //    os << format("\t> scatterer.reflectivity.shape = [{}, {}]",
    //                 1, scatterer.reflectivity.size()) ;

    //    // returning os
    //    return os;
    // }

    // // copy constructor from a pointer
    // ScattererClass(ScattererClass* scatterers){

    //    // copying the values
    //    this->coordinates = scatterers->coordinates;
    //    this->reflectivity = scatterers->reflectivity;
    // }

};
```

### 8.1.2 Class: Transmitter

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

```
// // 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(DATATYPE).to(DEVICE);
//        torch::Tensor absolute_elevation_of_transmitter = pitch + torch::tensor({this->elevation_angle}).to(DATATYPE).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(DATATYPE).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(DATATYPE).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;



//        /*
//        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);
```

```
//        torch::Tensor pointing_direction_spherical = fCart2Sph(this->pointing_direction);


//        // Calculating relative azimuths and radians
//        torch::Tensor relative_spherical = \
//            scatterers_spherical - pointing_direction_spherical;


//        // clearing some stuff up
//        scatterers_spherical.reset();
//        pointing_direction_spherical.reset();


//        // tensor corresponding to switch.
//        torch::Tensor tilt_angle_Tensor = \
//            torch::tensor({tilt_angle}).to(DATATYPE).to(DEVICE);

//        // calculating length of axes
//        torch::Tensor axis_a  = \
//            torch::tensor({
//                this->azimuthal_beamwidth / 2
//                }).to(DATATYPE).to(DEVICE);
//        torch::Tensor axis_b  = \
//            torch::tensor({
//                this->elevation_beamwidth / 2
//                }).to(DATATYPE).to(DEVICE);

//        // 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();


//        // 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;


//        // clearing
//        xcosa.reset(); ysina.reset(); xsina.reset(); ycosa.reset();


//        // 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});


//        // this is where histogram shadowing comes in (later)
//        if (ENABLE_RAYTRACING) {
//            rangeHistogramShadowing(scatterers);
//        }

//        // 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(DATATYPE);
//        elevation_centers = elevation_centers + torch::tensor({this->elevation_angle - \
//                                                            (this->elevation_beamwidth/2)}).to(DATATYPE); 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(DATATYPE);
//        torch::Tensor finalReflectivityBox = torch::zeros({numelevationcells, numazimuthcells}).to(DATATYPE); 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);

//          }
//      }
//  }




// };



template <typename T>
class TransmitterClass{
public:

    // physical/intrinsic properties
    std::vector<T>    location;             // location tensor
    std::vector<T>    pointing_direction;   // pointing direction

    // basic parameters
    std::vector<T>    Signal;               // transmitted signal (LFM)
    T                 azimuthal_angle;      // transmitter's azimuthal pointing direction
    T                 elevation_angle;      // transmitter's elevation pointing direction
    T                 azimuthal_beamwidth;  // azimuthal beamwidth of transmitter
    T                 elevation_beamwidth;  // elevation beamwidth of transmitter
    T                 range;                // a parameter used for spotlight mode.

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

    // shadowing properties
    int               azimuthQuantDensity;      // quantization of angles along the azimuth
    int               elevationQuantDensity;    // quantization of angles along the elevation
```

```
    T                 rangeQuantSize;          // range-cell size when shadowing
    T                 azimuthShadowThreshold;  // azimuth thresholding
    T                 elevationShadowThreshold; // elevation thresholding

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

    // constructor
    TransmitterClass() = default;

    // Deleting copy constructors/assignment
    TransmitterClass(const TransmitterClass& other)       = delete;
    TransmitterClass& operator=(TransmitterClass& other)  = delete;

    // Creating move-constructor and move-assignment
    TransmitterClass(TransmitterClass&& other)            = default;
    TransmitterClass& operator=(TransmitterClass&& other) =  default;

    /*==========================================================================
    Aim: Update pointing angle
    ----------------------------------------------------------------------------
    Note:
       > This function updates pointing angle based on AUV's pointing angle
       > for now, we're assuming no roll;
    --------------------------------------------------------------------------*/
    auto updatePointingAngle(std::vector<T> AUV_pointing_vector);

};



/*==========================================================================
Aim: Update pointing angle
----------------------------------------------------------------------------
Note:
   > This function updates pointing angle based on AUV's pointing angle
   > for now, we're assuming no roll;
--------------------------------------------------------------------------*/
template <typename T>
auto TransmitterClass<T>::updatePointingAngle(std::vector<T> AUV_pointing_vector)
{

    // calculate yaw and pitch
    auto   AUV_pointing_vector_spherical {svr::cart2sph(AUV_pointing_vector)};
    auto   yaw                          {AUV_pointing_vector_spherical[0]};
    auto   pitch                        {AUV_pointing_vector_spherical[1]};

    // calculating azimuth and elevation of transmitter object
    auto   absolute_azimuth_of_transmitter {yaw + this->azimuthal_angle};
    auto   absolute_elevation_of_transmitter {pitch + this->elevation_angle};

    // converting back to Cartesian
    auto   pointing_direction_spherical     {std::vector<T>(3, 0)};
    pointing_direction_spherical[0]         = absolute_azimuth_of_transmitter;
    pointing_direction_spherical[1]         = absolute_elevation_of_transmitter;
    pointing_direction_spherical[2]         = 1;
    this->pointing_direction                = svr::sph2cart(pointing_direction_spherical);
}
```

### 8.1.3 Class: Uniform Linear Array

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

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


// // class definitions
// #include "ScattererClass.h"
// #include "TransmitterClass.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 "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fBuffer2D.cpp"
// #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fConvolve1D.cpp"
// #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fCart2Sph.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 PRINTDOTS
//     #define PRINTDOTS
     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;                            // the new decimation factor
//    float post_decimation_sampling_frequency;         // the new sampling frequency
//    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
//    int num_buffer_zeros_per_frame; // number of zeros we're adding per frame to ensure no-rotation
//    torch::Tensor beamformedImage;  // the beamformed image
//    torch::Tensor cartesianImage;

//    // artificial acoustic-image related
//    torch::Tensor currentArtificalAcousticImage; // the acoustic image directly produced

//    // 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.to(DATATYPE);
//        }

//    // 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;
//        if (DEBUG_ULA) std::cout << "\t ULACLASS::init: line 136" << std::endl;

```

```
//        // status printing
//        if (DEBUG_ULA) {
//            std::cout << "\t\t ULAClass::init():: this->range_resolution = " \
//                      <<  this->range_resolution                          \
//                      <<  std::endl;
//            std::cout << "\t\t ULAClass::init():: this->range_cell_size = " \
//                      <<  this->range_cell_size                           \
//                      <<  std::endl;
//        }
//        if (DEBUG_ULA) std::cout << "\t ULACLASS::init: line 147" << 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;
//        if (DEBUG_ULA) std::cout << "\t ULACLASS::init: line 154" << std::endl;

//        // creating and storing the match-filter
//        this->nfdc_CreateMatchFilter(transmitterObj);
//        if (DEBUG_ULA) std::cout << "\t ULACLASS::init: line 158" << std::endl;
//    }1

//    // 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(                                    \
//                -1 * COMPLEX_1j * 2 * PI                        \
//                * (transmitterObj->fc/this->sampling_frequency) \
//                * basebanding_vector);
//        if (DEBUG_ULA) std::cout << "\t\t ULAClass::nfdc_createMatchFilter: line 176" << std::endl;

//        // multiplying the signal with the basebanding vector
//        torch::Tensor match_filter =                   \
//            torch::mul(transmitterObj->Signal,         \
//                       basebanding_vector);
//        if (DEBUG_ULA) std::cout << "\t\t ULAClass::nfdc_createMatchFilter: line 182" << std::endl;

//        // low-pass filtering to get the baseband signal
//        fConvolve1D(match_filter, this->lowpassFilterCoefficientsForDecimation);
//        if (DEBUG_ULA) std::cout << "\t\t ULAClass::nfdc_createMatchFilter: line 186" << std::endl;

//        // 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);
//        if (DEBUG_ULA) std::cout << "ULAClass::nfdc_createMatchFilter: line 197" << std::endl;

//        // 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, \
//                                                                     DATATYPE));
//        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(DATATYPE);
//        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";

//        //
//        torch::Tensor test = torch::mul(torch::tile(integer_array, {3, 1}).to(DATATYPE), \
//                                    torch::tile(this->sensorDirection, {1,
    this->num_sensors}).to(DATATYPE));
//        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(DATATYPE);


//        // 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 x-coordinates
//      slottedSensors =                                                \
//          torch::tile(slottedSensors,                                 \
//                      {1, slottedCoordinates_shape[1], 1});

//      // slotting the coordinate of the transmitter and duplicating along dimensions [0] and [1]
//      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).to(DATATYPE);

//      // substracting distance between relative fields
//      torch::Tensor distBetweenScatterersAndTransmitter =             \
//          torch::linalg_norm(slottedCoordinates - slotted_location,   \
//                             2, 2, true, torch::kFloat).to(DATATYPE);

//      // 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
//      #pragma omp parallel for
//      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 reflectivity-scaled impulse in there
//              this->signalMatrix.index_put_({where_to_place, sensor_index},              \
//                                            this->signalMatrix.index({where_to_place,      \
//                                                                      sensor_index}) +      \
//                                              scatterers->reflectivity.index({0, \
//                                                                              scatter_index}));
//          }
//      }



//      // // Adding pulses
//      // for(int sensor_index = 0; sensor_index < this->num_sensors; ++sensor_index){

//      //     // indices associated with current index
//      //     torch::Tensor tensor_containing_placing_indices = \
//      //         samplesOfFlight[sensor_index].to(torch::kInt);

//      //     // calculating histogram
//      //     auto uniqueOutputs = at::_unique(tensor_containing_placing_indices, false, true);
//      //     torch::Tensor bruh = std::get<1>(uniqueOutputs);
//      //     torch::Tensor uniqueValues = std::get<0>(uniqueOutputs).to(torch::kInt);
//      //     torch::Tensor uniqueCounts = torch::bincount(bruh).to(torch::kInt);

//      //     // placing values according to histogram
//      //     this->signalMatrix.index_put_({uniqueValues.to(torch::kLong), sensor_index}, \
//      //                                   uniqueCounts.to(DATATYPE));

//      // }

//      // Creating matrix out of 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()});


//      // returning
//      return;
//  }

//  /* ===========================================================================
//  Aim: Decimating basebanded-received 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);

//        // 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)});

//        // low-pass filtering the signal
//        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()});

//        // // Cutting off samples in the front.
//        // int cutoffpoint = lowpassfilter_impulseresponse.size(0) - 1;
//        // this->signalMatrix =                                            \
//        //    this->signalMatrix.index({                                   \
//        //        torch::indexing::Slice(cutoffpoint,                      \
//        //                            torch::indexing::None),              \
//        //        torch::indexing::Slice()                                 \
//        //    });

//        // building parameters for downsampling
//        int decimation_factor       = std::floor(this->sampling_frequency/transmitterObj->bandwidth);
//        this->decimation_factor      = decimation_factor;
//        this->post_decimation_sampling_frequency =                       \
//           this->sampling_frequency / this->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()});

//        // returning
//        return;
//    }

//    /* ==========================================================================
//    Aim: Match-filtering
//    ------------------------------------------------------------------------- */
//    void nfdc_matchFilterDecimatedSignal(){

//        // Creating a 2D matrix 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
//        fConvolveColumns(this->signalMatrix, \
//                       matchFilter2DMatrix);
```

```
//        // Trimming the signal to contain just the signals that make sense to us
//        int startingpoint = matchFilter2DMatrix.size(0) - 1;
//        this->signalMatrix =                                   \
//            this->signalMatrix.index({                         \
//                torch::indexing::Slice(startingpoint,          \
//                                       torch::indexing::None), \
//                torch::indexing::Slice()});

//        // // trimming the two ends of the signal
//        // int startingpoint = matchFilter2DMatrix.size(0) - 1;
//        // int endingpoint   = this->signalMatrix.size(0) \
//        //                     - matchFilter2DMatrix.size(0) \
//        //                     + 1;
//        // this->signalMatrix =                                   \
//        //    this->signalMatrix.index({                          \
//        //        torch::indexing::Slice(startingpoint,           \
//        //                               endingpoint), \
//        //        torch::indexing::Slice()});


//    }

//    /* =========================================================================
//    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);

//        // creating centers of range-cell centers
//        torch::Tensor range_centers = \
//            this->range_cell_size * \
//            torch::arange(0, number_of_range_cells) \
//            + this->range_cell_size/2;
//        this->range_centers = range_centers;

//        // creating discretized azimuth-centers
//        torch::Tensor azimuth_centers =                \
//            this->azimuth_cell_size *                  \
//            torch::arange(0, number_of_azimuths_to_image) \
//            + this->azimuth_cell_size/2;
//        this->azimuth_centers = azimuth_centers;
//        this->azimuth_centers = this->azimuth_centers - torch::mean(this->azimuth_centers);

//        // 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

//        // 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});

//        // creating x_m tensor
//        torch::Tensor sensorCoordinatesSlot = \
//            this->inter_element_spacing * \
//            torch::arange(0, this->num_sensors).reshape({
//                1, 1, this->num_sensors
//            }).to(DATATYPE);

//        sensorCoordinatesSlot = \
//            torch::tile(sensorCoordinatesSlot, \
//                        {range_grid.size(0),\
```

```
//                       range_grid.size(1),
//                       1});

//        // 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;
//        torch::Tensor sampleMatrix = timeMatrix * this->sampling_frequency;

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

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


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

//        // // calculating the buffer-zeros to add
//        // int num_buffer_zeros_per_frame = \
//        //    static_cast<float>(this->num_sensors - 1) * \
//        //    static_cast<float>(this->inter_element_spacing) * \
//        //    this->sampling_frequency /1500;

//        int num_buffer_zeros_per_frame =              \
//            std::ceil((this->num_sensors - 1) *        \
//                      this->inter_element_spacing *    \
//                      this->sampling_frequency         \
//                      / (1500 * this->decimation_factor));

//        // storing to class member
//        this->num_buffer_zeros_per_frame = \
//            num_buffer_zeros_per_frame;

//        // calculating the total frame-size
//        int total_frame_size = \
//            this->frame_size + this->num_buffer_zeros_per_frame;

//        // creating the multiplication matrix
//        torch::Tensor mulFFTMatrix = \
//            torch::linspace(0, \
//                          total_frame_size-1, \
//                          total_frame_size).reshape({1, \
//                                                   total_frame_size, \
//                                                   1}).to(DATATYPE); // creating an array 1,...,frame_size of shape [1,frame_size, 1];
//        mulFFTMatrix = \
//            torch::div(mulFFTMatrix, \
//                      torch::tensor(total_frame_size).to(DATATYPE)); // 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, total_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);
//           }
//       }

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

//   /* ================================================================================
//   Aim: 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");

//       // adding the batch-dimension
//       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() * this->frame_size);
//       int num_zeros_to_pad_signal_along_dimension_0 = \
//           ideal_length - this->signalMatrix.size(1);


//       // printing
//       if (DEBUG_ULA) PRINTSMALLLINE
//       if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->range_centers.numel()     = "<<this->range_centers.numel() <<std::endl;
//       if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->frame_size                = "<<this->frame_size             <<std::endl;
//       if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | ideal_length                    = "<<ideal_length                 <<std::endl;
//       if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->signalMatrix.size(1)      = "<<this->signalMatrix.size(1)   <<std::endl;
//       if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | num_zeros_to_pad_signal_along_dimension_0 = "<<num_zeros_to_pad_signal_along_dimension_0 <<std::endl;
//       if (DEBUG_ULA) PRINTSPACE

//       // appending or slicing based on the requirements
//       if (num_zeros_to_pad_signal_along_dimension_0 <= 0) {

//           // sending out a warning that slicing is going on
```

```
//          if (DEBUG_ULA) std::cerr <<"\t\t ULAClass::nfdc_beamforming | Please note that the signal matrix has been sliced. This could lead to loss of information"<<std::endl;

//          // slicing the signal matrix
//          if (DEBUG_ULA) PRINTSPACE
//          if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->signalMatrix.shape (before slicing) = "<< this->signalMatrix.sizes().vec() <<std::endl;
//          this->signalMatrix = \
//              this->signalMatrix.index({torch::indexing::Slice(), \
//                                   torch::indexing::Slice(0, ideal_length), \
//                                   torch::indexing::Slice()});
//          if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->signalMatrix.shape (after slicing) = "<< this->signalMatrix.sizes().vec() <<std::endl;
//          if (DEBUG_ULA) PRINTSPACE

//      }
//      else {
//          // creating a zero-filled tensor to append to signal matrix
//          torch::Tensor zero_tensor =                             \
//              torch::zeros({this->signalMatrix.size(0),           \
//                            num_zeros_to_pad_signal_along_dimension_0, \
//                            this->num_sensors}).to(DATATYPE);

//          // appending to signal matrix
//          this->signalMatrix       =                              \
//              torch::cat({this->signalMatrix, zero_tensor}, 1);
//      }

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


//      // add some zeros to the end of frames to accomodate delaying of signals.
//      torch::Tensor zero_filled_tensor =                  \
//          torch::zeros({this->signalMatrix.size(0),       \
//                       this->num_buffer_zeros_per_frame, \
//                       this->num_sensors}).to(DATATYPE);
//      this->signalMatrix =                                \
//          torch::cat({this->signalMatrix,                 \
//                      zero_filled_tensor}, 1);


//      // 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 to delay each of the frame accordingly
//      this->signalMatrix = torch::mul(this->signalMatrix, \
//                                      this->mulFFTMatrix);

//      // summing up the signals
//      // this->signalMatrix = torch::sum(this->signalMatrix, \
//      //                                 2,                 \
//      //                                 true);
//      this->signalMatrix = torch::sum(this->signalMatrix, \
//                                      2,                 \
//                                      false);


//      // printing some stuff
//      if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming: this->azimuth_centers.numel() = "<<this->azimuth_centers.numel() <<std::endl;
//      if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming: this->range_centers.numel() = "<<this->range_centers.numel() <<std::endl;
//      if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming: total number             = "<<this->range_centers.numel() * this->azimuth_centers.numel() <<std::endl;
//      if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming: this->signalMatrix.sizes().vec() = "<<this->signalMatrix.sizes().vec() <<std::endl;

//      // creating a tensor to store the final image
//      torch::Tensor finalImage = \
//          torch::zeros({this->frame_size * this->range_centers.numel(), \
//                       this->azimuth_centers.numel()}).to(torch::kComplexFloat);
```

```
//        // creating a loop to assign values
//        for(int range_index = 0; range_index < this->range_centers.numel(); ++range_index){
//            for(int angle_index = 0; angle_index < this->azimuth_centers.numel(); ++angle_index){

//                // getting row index
//                int rowindex = \
//                    angle_index * this->range_centers.numel() \
//                    + range_index;

//                // getting the strip to store
//                torch::Tensor strip = \
//                    this->signalMatrix.index({rowindex, \
//                                              torch::indexing::Slice()});

//                // taking just the first few values
//                strip = strip.index({torch::indexing::Slice(0, this->frame_size)});

//                // placing the strips on the image
//                finalImage.index_put_({\
//                    torch::indexing::Slice((range_index)*this->frame_size, \
//                                           (range_index+1)*this->frame_size), \
//                                          angle_index}, \
//                                          strip);

//            }
//        }

//        // saving the image
//        this->beamformedImage = finalImage;



//        // converting image from polar to cartesian
//        nfdc_PolarToCartesian();


//    }

//    /* =========================================================================
//    Aim: Converting Polar Image to Cartesian
//    .........................................................................
//    Note:
//        > For now, we're assuming that the r value is one.
//    ---------------------------------------------------------------------- */
//    void nfdc_PolarToCartesian(){


//        // deciding image dimensions
//        int num_pixels_width  = 128;
//        int num_pixels_height = 128;


//        // creating query points
//        torch::Tensor max_right =                                       \
//            torch::cos(                                                 \
//                torch::max(                                             \
//                    this->azimuth_centers                               \
//                    - torch::mean(this->azimuth_centers)                \
//                    + torch::tensor({90}).to(DATATYPE))             \
//                * PI/180);
//        torch::Tensor max_left  =                                       \
//            torch::cos(                                                 \
//                torch::min(this->azimuth_centers                        \
//                        - torch::mean(this->azimuth_centers)            \
//                        + torch::tensor({90}).to(DATATYPE))         \
//                * PI/180);
//        torch::Tensor max_top   = torch::tensor({1});
//        torch::Tensor max_bottom = torch::min(this->range_centers);


//        // creating query points along the x-dimension
//        torch::Tensor query_x =                    \
//            torch::linspace(                       \
```

```
//                max_left,                               \
//                max_right,                              \
//                num_pixels_width                        \
//                ).to(DATATYPE);

//        torch::Tensor query_y =                         \
//            torch::linspace(                            \
//                max_bottom.item<float>(),               \
//                max_top.item<float>(),                  \
//                num_pixels_height                       \
//                ).to(DATATYPE);


//        // converting original coordinates to their corresponding cartesian
//        float delta_r = 1/static_cast<float>(this->beamformedImage.size(0));
//        float delta_azimuth =                           \
//            torch::abs(                                 \
//                this->azimuth_centers.index({1})        \
//                - this->azimuth_centers.index({0})      \
//                ).item<float>();



//        // getting query points
//        torch::Tensor range_values = \
//            torch::linspace(                            \
//                delta_r,                                \
//                this->beamformedImage.size(0) * delta_r, \
//                this->beamformedImage.size(0)           \
//                ).to(DATATYPE);
//        range_values = \
//            range_values.reshape({range_values.numel(), 1});
//        range_values = \
//            torch::tile(range_values, \
//                        {1, this->azimuth_centers.numel()});


//        // getting angle-values
//        torch::Tensor angle_values =                    \
//            this->azimuth_centers                       \
//            - torch::mean(this->azimuth_centers)        \
//            + torch::tensor({90});
//        angle_values =                                  \
//            torch::tile(                                \
//                angle_values,                           \
//                {this->beamformedImage.size(0), 1});


//        // converting to cartesian original points
//        torch::Tensor query_original_x = \
//            range_values * torch::cos(angle_values * PI/180);
//        torch::Tensor query_original_y = \
//            range_values * torch::sin(angle_values * PI/180);



//        // converting points to vector 2D format
//        torch::Tensor query_source =                                      \
//            torch::cat({                                                  \
//                query_original_x.reshape({1, query_original_x.numel()}),  \
//                query_original_y.reshape({1, query_original_y.numel()})}, \
//                0);


//        // converting reflectivity to corresponding 2D format
//        torch::Tensor reflectivity_vectors = \
//            this->beamformedImage.reshape({1, this->beamformedImage.numel()});


//        // creating image
//        torch::Tensor cartesianImageLocal =             \
//            torch::zeros(                               \
//                {num_pixels_height,                     \
//                 num_pixels_width}                      \
//                 ).to(torch::kComplexFloat);
```

```

//        /*
//        Next Aim: start interpolating the points on the uniform grid.
//        */
//        #pragma omp parallel for
//        for(int x_index = 0; x_index < query_x.numel(); ++x_index){
//            // if(DEBUG_ULA) std::cout << "\t\t\t x_index = " << x_index << " ";
//            #pragma omp parallel for
//            for(int y_index = 0; y_index < query_y.numel(); ++y_index){
//                // if(DEBUG_ULA) if(y_index%16 == 0) std::cout<<".";

//                // getting current values
//                torch::Tensor current_x = query_x.index({x_index}).reshape({1, 1});
//                torch::Tensor current_y = query_y.index({y_index}).reshape({1, 1});




//                // getting the query value
//                torch::Tensor query_vector = torch::cat({current_x, current_y}, 0);


//                // copying the query source
//                torch::Tensor query_source_relative = query_source;
//                query_source_relative = query_source_relative - query_vector;


//                // subsetting using absolute values and masks
//                float threshold = delta_r * 10;
//                // PRINTDOTS
//                auto mask_row = \
//                    torch::abs(query_source_relative[0]) <= threshold;
//                auto mask_col = \
//                    torch::abs(query_source_relative[1]) <= threshold;
//                auto mask_together = torch::mul(mask_row, mask_col);




//                // calculating number of points in threshold neighbourhood
//                int num_points_in_threshold_neighbourhood = \
//                    torch::sum(mask_together).item<int>();
//                if (num_points_in_threshold_neighbourhood == 0){
//                    continue;
//                }



//                // subsetting points in neighbourhood
//                torch::Tensor PointsInNeighbourhood =   \
//                    query_source_relative.index({
//                        torch::indexing::Slice(),       \
//                        mask_together});
//                torch::Tensor ReflectivitiesInNeighbourhood = \
//                    reflectivity_vectors.index({torch::indexing::Slice(), mask_together});


//                // finding the distance between the points
//                torch::Tensor relativeDistances = \
//                    torch::linalg_norm(PointsInNeighbourhood, \
//                                       2, 0, true, \
//                                       torch::kFloat).to(DATATYPE);


//                // calculating weighing factor
//                torch::Tensor weighingFactor =                              \
//                    torch::nn::functional::softmax(                         \
//                        torch::max(relativeDistances)- relativeDistances,  \
//                        torch::nn::functional::SoftmaxFuncOptions(1));


//                // combining intensities using distances
//                torch::Tensor finalIntensity = \
//                    torch::sum(
//                        torch::mul(weighingFactor, \
```

```
//                               ReflectivitiesInNeighbourhood));

//                // assigning values
//                cartesianImageLocal.index_put_({x_index, y_index}, finalIntensity);

//            }
//            // std::cout<<std::endl;
//        }

//        // saving to member function
//        this->cartesianImage = cartesianImageLocal;

//    }

//    /* ==========================================================================
//    Aim: create acoustic image directly
//    -------------------------------------------------------------------------- */
//    void nfdc_createAcousticImage(ScattererClass* scatterers, \
//                                  TransmitterClass* transmitterObj){

//        // first we ensure that the scattersers are in our frame of reference
//        scatterers->coordinates = scatterers->coordinates - this->location;

//        // finding the spherical coordinates of the scatterers
//        torch::Tensor scatterers_spherical = fCart2Sph(scatterers->coordinates);

//        // note that its not precisely projection. its rotation. So the original lengths must be maintained. but thats easy since the operation of putting th eelevation to be zero works just fine.
//        scatterers_spherical.index_put_({1, torch::indexing::Slice()}, 0);

//        // converting the points back to cartesian
//        torch::Tensor scatterers_acoustic_cartesian = fSph2Cart(scatterers_spherical);

//        // removing the z-dimension
//        scatterers_acoustic_cartesian = \
//            scatterers_acoustic_cartesian.index({torch::indexing::Slice(0, 2, 1), \
//                                                 torch::indexing::Slice()});

//        // deciding image dimensions
//        int num_pixels_x = 512;
//        int num_pixels_y = 512;
//        torch::Tensor acousticImage =                  \
//            torch::zeros({num_pixels_x,                \
//                          num_pixels_y}).to(DATATYPE);

//        // finding the max and min values
//        torch::Tensor min_x   = torch::min(scatterers_acoustic_cartesian[0]);
//        torch::Tensor max_x   = torch::max(scatterers_acoustic_cartesian[0]);
//        torch::Tensor min_y   = torch::min(scatterers_acoustic_cartesian[1]);
//        torch::Tensor max_y   = torch::max(scatterers_acoustic_cartesian[1]);

//        // creating query grids
//        torch::Tensor query_x = torch::linspace(0, 1, num_pixels_x);
//        torch::Tensor query_y = torch::linspace(0, 1, num_pixels_y);

//        // scaling it up to image max-point spread
//        query_x           = min_x + (max_x - min_x) * query_x;
//        query_y           = min_y + (max_y - min_y) * query_y;
//        float delta_queryx = (query_x[1] - query_x[0]).item<float>();
//        float delta_queryy = (query_y[1] - query_y[0]).item<float>();

//        // creating a mesh-grid
//        auto queryMeshGrid = torch::meshgrid({query_x, query_y}, "ij");
//        query_x = queryMeshGrid[0].reshape({1, queryMeshGrid[0].numel()});
//        query_y = queryMeshGrid[1].reshape({1, queryMeshGrid[1].numel()});;
//        torch::Tensor queryMatrix = torch::cat({query_x, query_y}, 0);

//        // printing shapes
//        if(DEBUG_ULA) PRINTSMALLLINE
//        if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: query_x.shape = "<<query_x.sizes().vec()<<std::endl;
//        if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: query_y.shape = "<<query_y.sizes().vec()<<std::endl;
//        if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: queryMatrix.shape = "<<queryMatrix.sizes().vec()<<std::endl;
```

```
//        // setting up threshold values
//        float threshold_value =      \
//            std::min(delta_queryx,    \
//                     delta_queryy); if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 711"<<std::endl;

//        // putting a loop through the whole thing
//        for(int i = 0; i<queryMatrix[0].numel(); ++i){
//            // for each element in the query matrix
//            if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 716"<<std::endl;

//            // calculating relative position of all the points
//            torch::Tensor relativeCoordinates = \
//                scatterers_acoustic_cartesian - \
//                queryMatrix.index({torch::indexing::Slice(), i}).reshape({2, 1}); if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 720"<<std::endl;

//            // calculating distances between all the points and the query point
//            torch::Tensor relativeDistances = \
//                torch::linalg_norm(relativeCoordinates, \
//                                   1, 0, true, \
//                                   DATATYPE);if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 727"<<std::endl;
//            // finding points that are within the threshold
//            torch::Tensor conditionMeetingPoints = \
//                relativeDistances.squeeze() <= threshold_value;if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 729"<<std::endl;

//            // subsetting the points in the neighbourhood
//            if(torch::sum(conditionMeetingPoints).item<float>() == 0){

//                // continuing implementation if there are no points in the neighbourhood
//                continue; if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 735"<<std::endl;
//            }
//            else{
//                // creating mask for points in the neighbourhood
//                auto mask = (conditionMeetingPoints == 1);if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 739"<<std::endl;

//                // subsetting relative distances in the neighbourhood
//                torch::Tensor distanceInTheNeighbourhood = \
//                    relativeDistances.index({torch::indexing::Slice(), mask});if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 743"<<std::endl;

//                // subsetting reflectivity of points in the neighbourhood
//                torch::Tensor reflectivityInTheNeighbourhood = \
//                    scatterers->reflectivity.index({torch::indexing::Slice(), mask});if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 747"<<std::endl;

//                // assigning intensity as a function of distance and reflectivity
//                torch::Tensor reflectivityAssignment =                    \
//                    torch::mul(torch::exp(-distanceInTheNeighbourhood),   \
//                               reflectivityInTheNeighbourhood);if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 752"<<std::endl;
//                reflectivityAssignment = \
//                    torch::sum(reflectivityAssignment);if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 754"<<std::endl;

//                // assigning this value to the image pixel intensity
//                int pixel_position_x = i%num_pixels_x;
//                int pixel_position_y = std::floor(i/num_pixels_x);
//                acousticImage.index_put_({pixel_position_x, \
//                                         pixel_position_y}, \
//                                        reflectivityAssignment.item<float>());if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 761"<<std::endl;
//            }

//        }

//        // storing the acoustic-image to the member
//        this->currentArtificalAcousticImage = acousticImage;

//        // // saving the torch::tensor
```

```
//        // torch::save(acousticImage, \
//        //           "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/Assets/acoustic_image.pt");



//        // // bringing it back to the original coordinates
//        // scatterers->coordinates = scatterers->coordinates + this->location;
//     }



// };


template <typename T>
class ULAClass
{
public:
    // intrinsic parameters
    int                             num_sensors;                            // number of sensors
    T                               inter_element_spacing;                  // space between sensors
    std::vector<std::vector<T>>     coordinates;                            // coordinates of each sensor
    T                               sampling_frequency;                     // sampling frequency of the sensors
    T                               recording_period;                       // recording period of the ULA
    std::vector<T>                  location;                               // location of first coordinate

    // derived
    std::vector<T>                  sensor_direction;
    std::vector<std::vector<T>>     signalMatrix;

    // decimation related
    int               decimation_factor;                          // the new decimation factor
    T                 post_decimation_sampling_frequency;         // the new sampling frequency
    std::vector<T>    lowpass_filter_coefficients_for_decimation; // filter-coefficients for filtering

    // imaging related
    T   range_resolution;           // theoretical range-resolution = $\frac{c}{2B}$
    T   azimuthal_resolution;       // theoretical azimuth-resolution = $\frac{\lambda}{(N-1)*inter-element-distance}$
    T   range_cell_size;            // the range-cell quanta we're choosing for efficiency trade-off
    T   azimuth_cell_size;          // the azimuth quanta we're choosing
    std::vector<T>  azimuth_centers; // tensor containing the azimuth centeres
    std::vector<T>  range_centers;  // tensor containing the range-centers
    int  frame_size;                // the frame-size corresponding to a range cell in a decimated signal matrix

    std::vector<std::vector<complex<T>>> mulFFTMatrix; // the matrix containing the delays for each-element as a slot
    std::vector<complex<T>>             matchFilter;  // torch tensor containing the match-filter
    int    num_buffer_zeros_per_frame;                // number of zeros we're adding per frame to ensure no-rotation
    std::vector<std::vector<T>> beamformedImage;      // the beamformed image
    std::vector<std::vector<T>> cartesianImage;       // the cartesian version of beamformed image

    // Artificial acoustic-image related
    std::vector<std::vector<T>> currentArtificialAcousticImage; // acoustic image directly produced


    // Basic Constructor
    ULAClass() = default;

    // constructor
    ULAClass(const int    num_sensors_arg,
             const auto   inter_element_spacing_arg,
             const auto&  coordinates_arg,
             const auto&  sampling_frequency_arg,
             const auto&  recording_period_arg,
             const auto&  location_arg,
             const auto&  signalMatrix_arg,
             const auto&  lowpass_filter_coefficients_for_decimation_arg):
                num_sensors(num_sensors_arg),
                inter_element_spacing(inter_element_spacing_arg),
                coordinates(std::move(coordinates_arg)),
                sampling_frequency(sampling_frequency_arg),
                recording_period(recording_period_arg),
```

```
                location(std::move(location_arg)),
                signalMatrix(std::move(signalMatrix_arg)),
                lowpass_filter_coefficients_for_decimation(std::move(lowpass_filter_coefficients_for_decimation_arg))
    {

        // calculating ULA direction
        sensor_direction = std::vector<T>{coordinates[1][0] - coordinates[0][0],
                                          coordinates[1][1] - coordinates[0][1],
                                          coordinates[1][2] - coordinates[0][2]};

        // normalizing
        auto    norm_value_temp   {std::inner_product(sensor_direction.begin(), sensor_direction.end(),
                                                      sensor_direction.begin(),
                                                      0.00)};

        // dividing
        if (norm_value_temp != 0) {sensor_direction =  sensor_direction / norm_value_temp;}

    }

    // deleting copy constructor/assignment
    // ULAClass(const ULAClass& other)                   = delete;
    // ULAClass& operator=(const   ULAClass&  other)     = delete;


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

    /* ========================================================================
    Aim: Init
    ------------------------------------------------------------------------ */
    void   init(TransmitterClass<T>&   transmitterObj);

    /* ========================================================================
    Aim: Creating match-filter
    ------------------------------------------------------------------------ */
    void   nfdc_CreateMatchFilter(TransmitterClass<T>& transmitterObj);

};
// ==========================================================================================
template <typename T>
void ULAClass<T>::buildCoordinatesBasedOnLocation()
{

    // length-normalizing sensor-direction
    this->sensor_direction   =   this->sensor_direction / norm(this->sensor_direction);

    // multiply with inter-element distance
    this->sensor_direction   =   this->sensor_direction * this->inter_element_spacing;

    // create integer array
    auto   integer_array     {linspace<T>(0, this->num_sensors-1, this->num_sensors)};
    auto   test    {svr::tile(integer_array, {3,1}) * \
                    svr::tile(transpose(this->sensor_direction),
                              {1, static_cast<std::size_t>(this->num_sensors)})};
    this->coordinates =   this->location + test;
}

/* ========================================================================
Aim: Init
------------------------------------------------------------------------ */
template <typename T>
void   ULAClass<T>::init(TransmitterClass<T>&  transmitterObj)
{

    // calculating range-related parameters
    this->range_resolution   =   1500.00/(2 * transmitterObj.fc);
    this->range_cell_size    =   40  *   this->range_resolution;

    // calculating azimuth-related parameters
    this->azimuthal_resolution = (1500.00 / transmitterObj.fc) / \
                                 (this->num_sensors - 1) * (this->inter_element_spacing);
    this->azimuth_cell_size  =   2   *   this->azimuthal_resolution;

    // creating and storing match-filter
```

```
    this->nfdc_CreateMatchFilter(transmitterObj);

}

/* ==========================================================================
Aim: Creating match-filter
-------------------------------------------------------------------------- */
template <typename T>
void   ULAClass<T>::nfdc_CreateMatchFilter(TransmitterClass<T>& transmitterObj)
{
    svr::Timer timer("nfdc_CreateMatchFilter");

    // creating matrix for basebanding signal
    auto   linspace00           {linspace(0,
                                          transmitterObj.Signal.size()-1,
                                          transmitterObj.Signal.size())};
    auto   basebanding_vector   {linspace00 * \
                                 exp(-1.00 * 1i * 2.00 * std::numbers::pi * \
                                     (transmitterObj.fc / this->sampling_frequency)*\
                                     linspace00)};

    // multiplying signal with basebanding signal
    auto   match_filter      {transmitterObj.Signal * basebanding_vector};

    // low-pass filtering with baseband signal to obtain baseband signal
    match_filter   =   svr::conv1D(match_filter,
                                this->lowpass_filter_coefficients_for_decimation);

    // creating sampling-indices
    int    decimation_factor {static_cast<int>(std::floor(
        (static_cast<T>(this->sampling_frequency)/2.00) / \
        (static_cast<T>(transmitterObj.bandwidth)/2.00))
    )};
    int    final_num_samples {static_cast<int>(std::ceil(
        static_cast<T>(match_filter.size())/           \
        static_cast<T>(decimation_factor)
    ))};
    auto   sampling_indices  {
        linspace(1,
                 (final_num_samples - 1) * decimation_factor,
                 final_num_samples)
    };

    // sampling the signal
    match_filter   =   svr::index(match_filter, sampling_indices);

    // taking conjugate and flipping the signal
    match_filter   =   svr::fliplr(  match_filter);
    match_filter   =   svr::conj(    match_filter);

    // storing the match-filter to the class member
    this->matchFilter =   std::move(match_filter);
}
```

### 8.1.4 Class: Autonomous Underwater Vehicle

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

---

```


// #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;
//     save(SeafloorScatter_gt.coordinates,
    "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt.pt");
//     save(SeafloorScatter_gt.reflectivity,
    "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_gt_reflectivity.pt");

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

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

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

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


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

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

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

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


// // hash-defines
// #define PI                             3.14159265
// #define DEBUGMODE_AUV                  false
// #define SAVE_SIGNAL_MATRIX             true
// #define SAVE_DECIMATED_SIGNAL_MATRIX   true
// #define SAVE_MATCHFILTERED_SIGNAL_MATRIX true

// class AUVClass{
// public:
//    // Intrinsic attributes
//    Tensor location;                // location of vessel
//    Tensor velocity;                // current speed of the vessel [a vector]
//    Tensor acceleration;            // current acceleration of vessel [a vector]
//    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
//    Tensor signalMatrix_1;          // matrix containing the signals obtained from ULA_1
//    Tensor largeSignalMatrix_1;     // matrix holding signal of synthetic aperture
//    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
//    Tensor absolute_coords_patch_cart; // cartesian coordinates of patch

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








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

//        // call sync-component attributes
//        this->syncComponentAttributes();
//        if (DEBUGMODE_AUV) cout << "AUVCLass::init: line 128" << endl;
```

```
//        // 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);
//        if (DEBUGMODE_AUV) cout << "AUVCLass::init: line 134" << endl;


//        // precomputing delay-matrices for the ULA-class
//        thread ULA_fls_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
//                                            &this->ULA_fls,                         \
//                                            &this->transmitter_fls);
//        thread ULA_port_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
//                                             &this->ULA_port,                       \
//                                             &this->transmitter_port);
//        thread ULA_starboard_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
//                                                  &this->ULA_starboard,             \
//                                                  &this->transmitter_starboard);
//        if (DEBUGMODE_AUV) cout << "AUVCLass::init: line 145" << endl;

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

//    }



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

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

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



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

//        // updating ULA attributes
//        if(DEBUGMODE_AUV) 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
//        Tensor ula_fls_sensor_direction_spherical = \
//            fCart2Sph(this->pointing_direction);         // spherical coords
//        ula_fls_sensor_direction_spherical[0]          = \
//            ula_fls_sensor_direction_spherical[0] - 90;
//        Tensor ula_fls_sensor_direction_cart     = \
//            fSph2Cart(ula_fls_sensor_direction_spherical);

//        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

//        // // calling the function to update the arguments
//        // this->ULA_fls.buildCoordinatesBasedOnLocation();  if(DEBUGMODE_AUV) cout<<"\t AUVClass: line 109
```

```
\n";
//        // this->ULA_port.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) cout<<"\t AUVClass: line 111 \n";
//        // this->ULA_starboard.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) 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;

//        // 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);
//    }



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


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

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

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

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

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

//        Tensor yawCorrectionMatrix = \
//            tensor({cos(azimuth_correction_radians).item<float>(), \
//                         cos(tensor({90}).to(DATATYPE).to(DEVICE)*PI/180 + azimuth_correction_radians).item<float>(), \
//                         (float)0,                                                  \
//                         sin(azimuth_correction_radians).item<float>(), \
//                         sin(tensor({90}).to(DATATYPE).to(DEVICE)*PI/180 + azimuth_correction_radians).item<float>(), \
//                         (float)0,                                                  \
//                         (float)0,                                                  \
//                         (float)0,                                                  \
//                         (float)1}).reshape({3,3}).to(DATATYPE).to(DEVICE);

//        // returning the matrix
//        return yawCorrectionMatrix;
//    }

//    // pitch-correction matrix
//    Tensor createPitchCorrectionMatrix(Tensor pointing_direction_spherical, \
```

```
//                                          float target_elevation_deg){

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

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

//         // returning the matrix
//         return pitchCorrectionMatrix;
//     }

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

//         // printing status
//         cout << "\t AUVClass::simulateSignal: Began Signal Simulation" << endl;

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

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

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


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


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

```
//                                  &this->transmitter_port);
//        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


//    }

//    // Imaging Function
//    /* ========================================================================
//    ------------------------------------------------------------------------ */
//    void image(){

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

//        // saving the decimated signal
//        if (SAVE_DECIMATED_SIGNAL_MATRIX) {
//            cout << "\t AUVClass::image: saving decimated signal matrix" \
//                      << endl;
//            save(this->ULA_fls.signalMatrix, \
//
    "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/decimated_signalMatrix_fls.pt");
//            save(this->ULA_port.signalMatrix, \
//
    "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/decimated_signalMatrix_port.pt");
//            save(this->ULA_starboard.signalMatrix, \
//
    "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/decimated_signalMatrix_starboard.pt");
//        }

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


//        // saving the decimated signal
//        if (SAVE_MATCHFILTERED_SIGNAL_MATRIX) {

//            // saving the tensors
//            cout << "\t AUVClass::image: saving match-filtered signal matrix" \
//                      << endl;
//            save(this->ULA_fls.signalMatrix, \
```

```
//
    "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/matchfiltered_signalMatrix_fls.pt");
//            save(this->ULA_port.signalMatrix, \
//
    "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/matchfiltered_signalMatrix_port.pt");
//            save(this->ULA_starboard.signalMatrix, \
//
    "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/matchfiltered_signalMatrix_starboard.pt");

//            // running python-script

//        }


//        // performing the beamforming
//        thread ULA_fls_beamforming_t(&ULAClass::nfdc_beamforming,   \
//                                     &this->ULA_fls,                  \
//                                     &this->transmitter_fls);
//        thread ULA_port_beamforming_t(&ULAClass::nfdc_beamforming,  \
//                                      &this->ULA_port,                 \
//                                      &this->transmitter_port);
//        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: directly create acoustic image
//    ------------------------------------------------------------------------- */
//    void createAcousticImage(ScattererClass* scatterers){

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

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

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

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


//        // asking the ULAs to directly create acoustic images
//        thread ULA_fls_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, this->ULA_fls, \
//                                         &scatterer_fls, &this->transmitter_fls);
//        thread ULA_port_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, &this->ULA_port, \
//                                         &scatterer_port, &this->transmitter_port);
//        thread ULA_starboard_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, &this->ULA_starboard, \
//                                               &scatterer_starboard, &this->transmitter_starboard);

//        // joining the threads back
//        ULA_fls_acoustic_image_t.join(   );
//        ULA_port_acoustic_image_t.join( );
//        ULA_starboard_acoustic_image_t.join();

//    }


// };
























// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0000,
// // 0.0001,
```

```
// // 0.0001,
// // 0.0002,
// // 0.0003,
// // 0.0006,
// // 0.0009,
// // 0.0014,
// // 0.0022, 0.0032, 0.0047, 0.0066, 0.0092, 0.0126, 0.0168, 0.0219, 0.0281, 0.0352, 0.0432, 0.0518, 0.0609,
    0.0700, 0.0786, 0.0861, 0.0921, 0.0958, 0.0969, 0.0950, 0.0903, 0.0833, 0.0755, 0.0694, 0.0693, 0.0825,
    0.1206












template <typename T>
class  AUVClass{
public:

   // Intrinsic attributes
   std::vector<T>    location;            // location of vessel
   std::vector<T>    velocity;            // velocity of the vessel
   std::vector<T>    acceleration;        // acceleration of vessel
   std::vector<T>    pointing_direction;  // AUV's pointing direction

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

   // transmitters
   TransmitterClass<T>  transmitter_fls;        // transmitter for front-looking SONAR
   TransmitterClass<T>  transmitter_portside;   // mounted transmitter [obj of class, TransmitterClass]
   TransmitterClass<T>  transmitter_starboard;  // mounted transmitter [obj of class, TransmitterClass]

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

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

   // spotlight mode related
   std::vector<std::vector<T>>  absolute_coords_patch_cart;  // cartesian coordinates of patch

   // Synthetic Aperture Related
   std::vector<std::vector<T>>  ApertureSensorLocations;     // sensor locations of aperture

   // functions
   void syncComponentAttributes();
   void init();

};

/*==========================================================================
Aim: update attributes
--------------------------------------------------------------------------*/
template <typename T>
void AUVClass<T>::syncComponentAttributes()
{
   // updating locations of ULAs
   this->ULA_fls.location      = this->location;
   this->ULA_portside.location  = this->location;
   this->ULA_starboard.location = this->location;


```

```
    // updating pointing-direction of ULAs
    auto   ula_fls_sensor_direction_spherical  {svr::cart2sph(this->pointing_direction)};
    ula_fls_sensor_direction_spherical[0] -= 90;
    auto   ula_fls_sensor_direction_cart       {svr::sph2cart(ula_fls_sensor_direction_spherical)};

    this->ULA_fls.sensor_direction      =  ula_fls_sensor_direction_cart;
    this->ULA_portside.sensor_direction =  -1 *  this->pointing_direction;
    this->ULA_starboard.sensor_direction = -1 *  this->pointing_direction;


    // calling function to update argumentss
    this->ULA_fls.buildCoordinatesBasedOnLocation();
    this->ULA_portside.buildCoordinatesBasedOnLocation();
    this->ULA_starboard.buildCoordinatesBasedOnLocation();

    // updating transmitter location
    this->transmitter_fls.location      =  this->location;
    this->transmitter_portside.location =  this->location;
    this->transmitter_starboard.location = this->location;

    // updating transmitter pointing direction
    this->transmitter_fls.updatePointingAngle(     this->pointing_direction);
    this->transmitter_portside.updatePointingAngle( this->pointing_direction);
    this->transmitter_starboard.updatePointingAngle( this->pointing_direction);

}

/* ==========================================================================
Aim: Init
----------------------------------------------------------------------------*/
template <typename T>
void AUVClass<T>::init()
{
    // call sync-component attributes
    this->syncComponentAttributes();

    // initializing all the ULAs
    this->ULA_fls.init(      this->transmitter_fls);
    this->ULA_portside.init( this->transmitter_portside);
    this->ULA_starboard.init( this->transmitter_starboard);

    // pre-computing delay-matrices for ULA-class

}
```

## 8.2 Setup Scripts

### 8.2.1 Seafloor Setup

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

```
void fSeaFloorSetup(ScattererClass<double>& scatterers){

    // auto   save_files    {false};
    const  auto   save_files          {false};
    const  auto   hill_creation_flag   {true};

    // sea-floor bounds
    auto   bed_width     {100.00};
    auto   bed_length    {100.00};

    // creating tensors for coordinates and reflectivity
    vector<vector<double>>      box_coordinates;
    vector<double>              box_reflectivity;

    // scatter density
    auto   bed_width_density {static_cast<double>( 10.00)};
    auto   bed_length_density {static_cast<double>( 10.00)};

    // setting up coordinates
    auto   xpoints    {linspace<double>(0.00,
                                     bed_width,
                                     bed_width * bed_width_density)};
    auto   ypoints    {linspace<double>(0.00,
                                     bed_length,
                                     bed_length * bed_length_density)};
    if(save_files) fWriteVector(xpoints, "../csv-files/xpoints.csv");   // verified
    if(save_files) fWriteVector(ypoints, "../csv-files/ypoints.csv");   // verified

    // creating mesh
    auto [xgrid, ygrid] = meshgrid(std::move(xpoints), std::move(ypoints));
    if(save_files) fWriteMatrix(xgrid,  "../csv-files/xgrid.csv");      // verified
    if(save_files) fWriteMatrix(ygrid,  "../csv-files/ygrid.csv");      // verified

    // reshaping
    auto   X      {reshape(xgrid, xgrid.size()*xgrid[0].size())};
    auto   Y      {reshape(ygrid, ygrid.size()*ygrid[0].size())};
    if(save_files) fWriteVector(X,      "../csv-files/X.csv");          // verified
    if(save_files) fWriteVector(Y,      "../csv-files/Y.csv");          // verified

    // creating heights of scatterers
    if(hill_creation_flag){

        // setting up hill parameters
        auto   num_hills     {10};

        // setting up placement of hills
        auto   points2D            {concatenate<0>(X, Y)};               // verified
        auto   min2D               {min<1, double>(points2D)};           // verified
        auto   max2D               {max<1, double>(points2D)};           // verified
        auto   hill_2D_center      {min2D + \
                                    rand({2, num_hills}) * (max2D - min2D)}; // verified

        // setup: hill-dimensions
        auto   hill_dimensions_min  {transpose(vector<double>{5, 5, 2})};   // verified
        auto   hill_dimensions_max  {transpose(vector<double>{30, 30, 10})}; // verified
        auto   hill_dimensions      {hill_dimensions_min + \
                                    rand({3, num_hills}) * (hill_dimensions_max - hill_dimensions_min)};
                                                       // verified

        // function-call: hill-creation function
        fCreateHills(hill_2D_center,
                     hill_dimensions,
                     points2D);

        // setting up floor reflectivity
        auto   floorScatter_reflectivity    {std::vector<double>(Y.size(), 1.00)};
```

```

        // populating the values of the incoming argument
        scatterers.coordinates   = std::move(points2D);
        scatterers.reflectivity  = std::move(floorScatter_reflectivity);

    }
    else{

        // assigning flat heights
        auto   Z      {std::vector<double>(Y.size(), 0)};

        // setting up floor coordinates
        auto   floorScatter_coordinates     {concatenate<0>(X, Y, Z)};
        auto   floorScatter_reflectivity    {std::vector<double>(Y.size(), 1)};

        // populating the values of the incoming argument
        scatterers.coordinates   = std::move(floorScatter_coordinates);
        scatterers.reflectivity  = std::move(floorScatter_reflectivity);

    }
}
```

### 8.2.2 Transmitter Setup

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

```cpp
template <typename T>
void fTransmitterSetup(TransmitterClass<T>& transmitter_fls,
                       TransmitterClass<T>&  transmitter_portside,
                       TransmitterClass<T>&  transmitter_starboard)
{
    // Setting up transmitter
    T      sampling_frequency    {160e3};             // sampling frequency
    T      f1                    {50e3};              // first frequency of LFM
    T      f2                    {70e3};              // second frequency of LFM
    T      fc                    {(f1 + f2)/2.00};    // finding center-frequency
    T      bandwidth             {std::abs(f2 - f1)}; // bandwidth
    T      pulselength           {5e-2};              // time of recording

    // building LFM
    auto   timearray     {linspace<T>(0.00,
                                     pulselength,
                                     std::floor(pulselength * sampling_frequency))};
    auto   K             {f2 - f1/pulselength}; // calculating frequency-slope
    auto   Signal        {cos(2 * std::numbers::pi * \
                          (f1 + K*timearray) * \
                          timearray)};         // frequency at each time-step, with f1 = 0

    // Setting up transmitter
    auto   location                     {std::vector<T>(3, 0)};      // location of transmitter
    T      azimuthal_angle_fls          {0};                         // initial pointing direction
    T      azimuthal_angle_port         {90};                        // initial pointing direction
    T      azimuthal_angle_starboard    {-90};                       // initial pointing direction

    T      elevation_angle              {-60};                       // initial pointing direction

    T      azimuthal_beamwidth_fls        {20};                      // azimuthal beamwidth of the signal cone
    T      azimuthal_beamwidth_port       {20};                      // azimuthal beamwidth of the signal cone
    T      azimuthal_beamwidth_starboard  {20};                      // azimuthal beamwidth of the signal cone

    T      elevation_beamwidth_fls        {20};                      // elevation beamwidth of the signal cone
    T      elevation_beamwidth_port       {20};                      // elevation beamwidth of the signal cone
    T      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)
    T      rangeQuantSize               {10};  // the length of a cell (used for shadowing)

    T      azimuthShadowThreshold       {1};   // azimuth threshold   (in degrees)
    T      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_portside.location             = location;                 // Assigning location
    transmitter_portside.Signal               = Signal;                   // Assigning signal
    transmitter_portside.azimuthal_angle      = azimuthal_angle_port;     // assigning azimuth angle
    transmitter_portside.elevation_angle      = elevation_angle;          // assigning elevation angle
    transmitter_portside.azimuthal_beamwidth  = azimuthal_beamwidth_port; // assigning azimuth-beamwidth
    transmitter_portside.elevation_beamwidth  = elevation_beamwidth_port; // assigning elevation-beamwidth
    // updating quantization densities
    transmitter_portside.azimuthQuantDensity  = azimuthQuantDensity;      // assigning azimuth quant density
    transmitter_portside.elevationQuantDensity = elevationQuantDensity;    // assigning elevation quant
         density
    transmitter_portside.rangeQuantSize       = rangeQuantSize;           // assigning range-quantization
    transmitter_portside.azimuthShadowThreshold = azimuthShadowThreshold;  // azimuth-threshold in shadowing
    transmitter_portside.elevationShadowThreshold = elevationShadowThreshold; // elevation-threshold in
         shadowing
    // signal related
    transmitter_portside.f_low                = f1;                       // assigning lower frequency
    transmitter_portside.f_high               = f2;                       // assigning higher frequency
    transmitter_portside.fc                   = fc;                       // assigning center frequency
    transmitter_portside.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;       // assigning
         azimuth-quant-density
    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 ULA Setup

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

```
template <typename T>
void fULASetup(ULAClass<T>&  ula_fls,
               ULAClass<T>&   ula_portside,
               ULAClass<T>&   ula_starboard)
{
    // setting up ula
    auto    num_sensors               {static_cast<int>(64)};          // number of sensors
    T       sampling_frequency        {static_cast<T>(160e3)};         // sampling frequency
    T       inter_element_spacing     {1500/(2*sampling_frequency)};   // space between samples
    T       recording_period          {10e-2};                         // sampling-period

    // building the direction for the sensors
    auto    ULA_direction             {std::vector<T>({-1, 0, 0})};
    auto    ULA_direction_norm        {norm(ULA_direction)};
    if (ULA_direction_norm != 0)      {ULA_direction = ULA_direction/ULA_direction_norm;}
    ULA_direction         =           ULA_direction  *   inter_element_spacing;

    // building coordinates for sensors
    auto    ULA_coordinates           {transpose(ULA_direction) * \
                                       linspace<double>(0.00,
                                                        num_sensors -1,
                                                        num_sensors)};

    // coefficients of decimation filter
    auto    lowpassfiltercoefficients {std::vector<T>{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}};

    // 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.sensor_direction                       = ULA_direction;             // ULA direction
    ula_fls.lowpass_filter_coefficients_for_decimation = lowpassfiltercoefficients; // storing coefficients


    // assigning values
    ula_portside.num_sensors                            = num_sensors;               // assigning number of sensors
    ula_portside.inter_element_spacing                  = inter_element_spacing;     // assigning inter-element spacing
    ula_portside.coordinates                            = ULA_coordinates;           // assigning ULA coordinates
    ula_portside.sampling_frequency                     = sampling_frequency;        // assigning sampling frequencys
    ula_portside.recording_period                       = recording_period;          // assigning recording period
    ula_portside.sensor_direction                       = ULA_direction;             // ULA direction
    ula_portside.lowpass_filter_coefficients_for_decimation = lowpassfiltercoefficients; // storing coefficients


    // 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.sensor_direction                        = ULA_direction;             // ULA direction
    ula_starboard.lowpass_filter_coefficients_for_decimation = lowpassfiltercoefficients; // storing
        coefficients
}
```

### 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(DATATYPE).to(DEVICE); //
      starting location of AUV
//     torch::Tensor velocity        = torch::tensor({5,0, 0}).reshape({3,1}).to(DATATYPE).to(DEVICE); //
      starting velocity of AUV
//     torch::Tensor pointing_direction = torch::tensor({1,0, 0}).reshape({3,1}).to(DATATYPE).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
// }









// =======================================================
template <typename T>
void fAUVSetup(AUVClass<T>& auv) {

    // building properties for the auv
    auto    location              {std::vector<T>{0, 50, 30}};     // starting location of AUV
    auto    velocity              {std::vector<T>{5, 0, 0}};       // starting velocity of AUV
    auto    pointing_direction    {std::vector<T>{1, 0, 0}};       // pointing direction of AUV

    // assigning
    auv.location              = std::move(location);               // assigning location of auv
    auv.velocity              = std::move(velocity);               // assigning vector representing velocity
    auv.pointing_direction    = std::move(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, DATATYPE).to(DEVICE);
    torch::Tensor xysplat_lengths = torch::linalg_norm(xysplat, 2, 0, true, torch::kFloat).to(DATATYPE);
    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, DATATYPE).to(DEVICE);
    torch::Tensor rho_values    = torch::linalg_norm(cartesian_vector, \
                                                    2, 0, true, torch::kFloat).to(DATATYPE);
    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

```
namespace svr {
    // ========================================================================
    template <typename T>
    auto    cart2sph(const    std::vector<T> cartesian_vector){

        // splatting the point onto xy-plane
        auto    xysplat    {cartesian_vector};
        xysplat[2]     =   0;

        // finding splat lengths
        auto    xysplat_lengths   {norm(xysplat)};

        // finding azimuthal and elevation angles
        auto    azimuthal_angles  {svr::atan2(xysplat[1], xysplat[0]) *   180.00/std::numbers::pi};
        auto    elevation_angles  {svr::atan2(cartesian_vector[2], xysplat_lengths) * 180.00/std::numbers::pi};
        auto    rho_values        {norm(cartesian_vector)};

        // creating tensor to send back
        auto    spherical_vector  {std::vector<T>{azimuthal_angles,
                                                  elevation_angles,
                                                  rho_values}};

        // moving it back
        return std::move(spherical_vector);

    }
    // ========================================================================
    template <typename T>
    auto    sph2cart(const std::vector<T> spherical_vector){

        // creating cartesian vector
        auto    cartesian_vector  {std::vector<T>(spherical_vector.size(), 0)};

        // populating
        cartesian_vector[0]    =   spherical_vector[2] * \
                                   cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
                                   cos(spherical_vector[0] * std::numbers::pi / 180.00);
        cartesian_vector[1]    =   spherical_vector[2] * \
                                   cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
                                   sin(spherical_vector[0] * std::numbers::pi / 180.00);
        cartesian_vector[2]    =   spherical_vector[2] * \
                                   sin(spherical_vector[1] * std::numbers::pi / 180.00);

        // returning
        return std::move(cartesian_vector);
    }
}
```

### 8.3.3 Column-Wise Convolution

```
/* ====================================
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;

    // converting the two arguments to float since fft doesn'tw ork with halfs
    inputMatrix = inputMatrix.to(torch::kFloat);
    kernelMatrix = kernelMatrix.to(torch::kFloat);

    // 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;

    // bringing them back to the pipeline datatype
    kernelMatrix = kernelMatrix.to(DATATYPE);

    // over-riding the result with the input so that we can save memory
    inputMatrix = convolvedResult.to(DATATYPE); 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
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/packages.h"
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/config.h" // hash-defines
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/classes.h" // class defs
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/setupscripts.h" // setup-scripts
#include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/functions.h" // functions


// main-function
int main() {

    // Ensuring no-gradients are built
    NoGradGuard no_grad;

    // Builing Sea-floor
    ScattererClass SeafloorScatter;
    thread scatterThread_t(SeafloorSetup, \
                           ref(SeafloorScatter));

    // Building ULA
    ULAClass ula_fls, ula_port, ula_starboard;
    thread ulaThread_t(ULASetup, \
                       ref(ula_fls), \
                       ref(ula_port), \
                       ref(ula_starboard));

    // Building Transmitter
    TransmitterClass transmitter_fls, transmitter_port, transmitter_starboard;
    thread transmitterThread_t(TransmitterSetup,
                               ref(transmitter_fls),
                               ref(transmitter_port),
                               ref(transmitter_starboard));

    // recombining threads
    scatterThread_t.join();  // making the scattetr population thread join back
    ulaThread_t.join();      // making the ULA population thread join back
    transmitterThread_t.join(); // making the transmitter 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 data-structures required for processing
    auv.init();

    // mimicking movement
    int number_of_stophops = 4;
    // if (true) return 0;
    for(int i = 0; i<number_of_stophops; ++i){
```

```

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

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

        // making the deep copy
        ScattererClass SeafloorScatter = SeafloorScatter_deepcopy;

        // signal simulation
        auv.simulateSignal(SeafloorScatter);

        // saving simulated signal
        if (SAVETENSORS) {

            // saving the signal matrix tensors
            save(auv.ULA_fls.signalMatrix, \
                        "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_fls.pt");
            save(auv.ULA_port.signalMatrix, \
                        "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_port.pt");
            save(auv.ULA_starboard.signalMatrix, \
                        "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_starboard.pt");

            // running python script
            string script_to_run = \
                "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/Python/Plot_SignalMatrix.py";
            thread plotSignalMatrix_t(fRunSystemScriptInSeperateThread, \
                                      script_to_run);
            plotSignalMatrix_t.detach();

        }


        if (IMAGING_TOGGLE) {

            // creating image from signals
            auv.image();

            // saving the tensors
            if(SAVETENSORS){
                // saving the beamformed images
                save(auv.ULA_fls.beamformedImage, \
                            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/ULA_fls_image.pt");
                // save(auv.ULA_port.beamformedImage, \
                //             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/ULA_port_image.pt");
                // save(auv.ULA_starboard.beamformedImage, \
                //
                     "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/ULA_starboard_image.pt");

                // saving cartesian image
                save(auv.ULA_fls.cartesianImage, \
                            "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/ULA_fls_cartesianImage.pt");

                // // running python file
                // system("python
                     /Users/vrsreeganesh/Documents/GitHub/AUV/Code/Python/Plot_BeamformedImage.py");
                system("python /Users/vrsreeganesh/Documents/GitHub/AUV/Code/Python/Plot_cartesianImage.py");
            }
        }



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

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

    }


```

```
  // returning
  return 0;
}
```

# Chapter 9

# Reading

## 9.1 Primary Books

1.

## 9.2 Interesting Papers

# Chapter 10

# General Purpose Templated Functions

## 10.1 CSV File-Writes

```
// ===========================================================================
template <typename T>
void fWriteVector(const vector<T>&        inputvector,
                  const string&           filename){

    // opening a file
    std::ofstream fileobj(filename);
    if (!fileobj)  {return;}

    // writing the real parts in the first column and the imaginary parts int he second column
    if constexpr(std::is_same_v<T, std::complex<double>> ||
                 std::is_same_v<T, std::complex<float>> ||
                 std::is_same_v<T, std::complex<long double>>){
        for(int i = 0; i<inputvector.size(); ++i){
            // adding entry
            fileobj << inputvector[i].real() << "+" << inputvector[i].imag() << "i";

            // adding delimiter
            if(i!=inputvector.size()-1) {fileobj << ",";}
            else                        {fileobj << "\n";}
        }
    }
    else{
        for(int i = 0; i<inputvector.size(); ++i){
            fileobj << inputvector[i];
            if(i!=inputvector.size()-1) {fileobj << ",";}
            else                        {fileobj << "\n";}
        }
    }

    // return
    return;
}
// Matrix writing =============================================================
template <typename T>
auto fWriteMatrix(const std::vector<std::vector<T>> inputMatrix,
                  const string                    filename){

    // opening a file
    std::ofstream fileobj(filename);

    // writing
    if (fileobj){
        for(int i = 0; i<inputMatrix.size(); ++i){
            for(int j = 0; j<inputMatrix[0].size(); ++j){
                fileobj << inputMatrix[i][j];
                if (j!=inputMatrix[0].size()-1)  {fileobj << ",";}
                else                             {fileobj << "\n";}
```

```
            }
        }
    }
    else{
        cout << format("File-write to {} failed\n", filename);
    }

}

template <>
auto fWriteMatrix(const std::vector<std::vector<std::complex<double>>> inputMatrix,
                  const string                                      filename){

    // opening a file
    std::ofstream fileobj(filename);

    // writing
    if (fileobj){
        for(int i = 0; i<inputMatrix.size(); ++i){
            for(int j = 0; j<inputMatrix[0].size(); ++j){
                fileobj << inputMatrix[i][j].real() << "+" << inputMatrix[i][j].imag() << "i";
                if (j!=inputMatrix[0].size()-1) {fileobj << ",";}
                else                            {fileobj << "\n";}
            }
        }
    }
    else{
        cout << format("File-write to {} failed\n", filename);
    }
}
```

## 10.2 abs

```
// =============================================================================
// y = abs(vector)
template <typename T>
auto abs(const std::vector<T>&  input_vector)
{
    // creating canvas
    auto    canvas     {input_vector};

    // calculating abs
    std::transform(canvas.begin(),
                   canvas.end(),
                   canvas.begin(),
                   [](auto& argx){return std::abs(argx);});

    // returning
    return std::move(canvas);
}
// =============================================================================
// y = abs(matrix)
template <typename T>
auto abs(const std::vector<std::vector<T>>  input_matrix)
{
    // creating canvas
    auto    canvas     {input_matrix};

    // applying element-wise abs
    std::transform(input_matrix.begin(),
                   input_matrix.end(),
                   input_matrix.begin(),
                    [](auto& argx){return std::abs(argx);});

    // returning
    return std::move(canvas);
}
```

## 10.3 Boolean Comparators

```cpp
// ===========================================================================
template <typename T, typename U>
auto operator<(const  std::vector<T>&   input_vector,
               const   U                scalar)
{
    // creating canvas
    auto   canvas    {std::vector<bool>(input_vector.size())};

    // transforming
    std::transform(input_vector.begin(), input_vector.end(),
                   canvas.begin(),
                   [&scalar](const auto& argx){
                       return argx < static_cast<T>(scalar);
                   });

    // returning
    return std::move(canvas);
}
// ===========================================================================
template <typename T, typename U>
auto operator<=(const  std::vector<T>&   input_vector,
                const   U                scalar)
{
    // creating canvas
    auto   canvas    {std::vector<bool>(input_vector.size())};

    // transforming
    std::transform(input_vector.begin(), input_vector.end(),
                   canvas.begin(),
                   [&scalar](const auto& argx){
                       return argx <= static_cast<T>(scalar);
                   });

    // returning
    return std::move(canvas);
}
// ===========================================================================
template <typename T, typename U>
auto operator>(const  std::vector<T>&   input_vector,
               const   U                scalar)
{
    // creating canvas
    auto   canvas    {std::vector<bool>(input_vector.size())};

    // transforming
    std::transform(input_vector.begin(), input_vector.end(),
                   canvas.begin(),
                   [&scalar](const auto& argx){
                       return argx > static_cast<T>(scalar);
                   });

    // returning
    return std::move(canvas);
}
// ===========================================================================
template <typename T, typename U>
auto operator>=(const  std::vector<T>&   input_vector,
                const   U                scalar)
{
    // creating canvas
    auto   canvas    {std::vector<bool>(input_vector.size())};

    // transforming
    std::transform(input_vector.begin(), input_vector.end(),
                   canvas.begin(),
                   [&scalar](const auto& argx){
                       return argx >= static_cast<T>(scalar);
                   });

    // returning
    return std::move(canvas);
}
```

## 10.4 Concatenate Functions

```
// input = [vector, vector],
// output = [vector]
template <std::size_t axis, typename T>
auto concatenate(const std::vector<T>&  input_vector_A,
                 const std::vector<T>&   input_vector_B) -> std::enable_if_t<axis == 1, std::vector<T> >
{
    // creating canvas vector
    auto   num_elements  {input_vector_A.size() + input_vector_B.size()};
    auto   canvas        {std::vector<T>(num_elements, (T)0) };

    // filling up the canvas
    std::copy(input_vector_A.begin(), input_vector_A.end(),
              canvas.begin());
    std::copy(input_vector_B.begin(), input_vector_B.end(),
              canvas.begin()+input_vector_A.size());

    // moving it back
    return std::move(canvas);

}
// ==========================================================
// input = [vector, vector],
// output = [matrix]
template <std::size_t axis, typename T>
auto concatenate(const std::vector<T>&  input_vector_A,
                 const std::vector<T>&   input_vector_B) -> std::enable_if_t<axis == 0,
                        std::vector<std::vector<T>> >
{
    // throwing error dimensions
    if (input_vector_A.size() != input_vector_B.size())
        std::cerr << "concatenate:: incorrect dimensions \n";

    // creating canvas
    auto   canvas     {std::vector<std::vector<T>>(
        2, std::vector<T>(input_vector_A.size())
    )};

    // filling up the dimensions
    std::copy(input_vector_A.begin(), input_vector_A.end(), canvas[0].begin());
    std::copy(input_vector_B.begin(), input_vector_B.end(), canvas[1].begin());

    // moving it back
    return std::move(canvas);

}
// ==========================================================
// input = [vector, vector, vector],
// output = [matrix]
template <std::size_t axis, typename T>
auto concatenate(const std::vector<T>&  input_vector_A,
                 const std::vector<T>&   input_vector_B,
                 const std::vector<T>&   input_vector_C) -> std::enable_if_t<axis == 0,
                        std::vector<std::vector<T>> >
{
    // throwing error dimensions
    if (input_vector_A.size() != input_vector_B.size() ||
        input_vector_A.size() != input_vector_C.size())
        std::cerr << "concatenate:: incorrect dimensions \n";

    // creating canvas
    auto   canvas     {std::vector<std::vector<T>>(
        3, std::vector<T>(input_vector_A.size())
    )};

    // filling up the dimensions
    std::copy(input_vector_A.begin(), input_vector_A.end(), canvas[0].begin());
    std::copy(input_vector_B.begin(), input_vector_B.end(), canvas[1].begin());
    std::copy(input_vector_C.begin(), input_vector_C.end(), canvas[2].begin());

    // moving it back
    return std::move(canvas);

```

```
}
// =======================================================
// input = [matrix, vector],
// output = [matrix]
template <std::size_t axis, typename T>
auto concatenate(const std::vector<std::vector<T>>& input_matrix,
                 const std::vector<T>              input_vector) -> std::enable_if_t<axis == 0,
                      std::vector<std::vector<T>> >
{
    // creating canvas
    auto   canvas     {input_matrix};

    // adding to the canvas
    canvas.push_back(input_vector);

    // returning
    return std::move(canvas);
}
```

## 10.5 Conjugate

```
namespace  svr {
    // =====================================================================
    template   <typename T>
    auto    conj(const std::vector<T>&  input_vector)
    {
        // creating canvas
        auto    canvas     {std::vector<T>(input_vector.size())};

        // calculating conjugates
        std::for_each(canvas.begin(), canvas.end(),
                      [](auto& argx){argx = std::conj(argx);});

        // returning
        return std::move(canvas);
    }
}
```

## 10.6 Convolution

```
namespace  svr {
    // =====================================================================
    template   <typename T1, typename T2>
    auto    conv1D(const   std::vector<T1>&   input_vector_A,
                   const   std::vector<T2>&   input_vector_B)
    {
        // resulting type
        using  T3  = decltype(std::declval<T1>() * std::declval<T2>());

        // creating canvas
        auto    canvas_length     {input_vector_A.size() + input_vector_B.size() - 1};

        // calculating fft of two arrays
        auto    fft_A      {svr::fft(input_vector_A, canvas_length)};
        auto    fft_B      {svr::fft(input_vector_B, canvas_length)};

        // element-wise multiplying the two matrices
        auto    fft_AB     {fft_A *   fft_B};

        // finding inverse FFT
        auto    convolved_result  {ifft(fft_AB)};

        // returning
        return std::move(convolved_result);
    }

```

```
}
```

## 10.7 Coordinate Change

```
namespace svr {
    // ============================================================================
    template <typename T>
    auto    cart2sph(const   std::vector<T> cartesian_vector){

        // splatting the point onto xy-plane
        auto    xysplat    {cartesian_vector};
        xysplat[2]     =   0;

        // finding splat lengths
        auto    xysplat_lengths   {norm(xysplat)};

        // finding azimuthal and elevation angles
        auto    azimuthal_angles  {svr::atan2(xysplat[1], xysplat[0]) *   180.00/std::numbers::pi};
        auto    elevation_angles  {svr::atan2(cartesian_vector[2], xysplat_lengths) * 180.00/std::numbers::pi};
        auto    rho_values        {norm(cartesian_vector)};

        // creating tensor to send back
        auto    spherical_vector  {std::vector<T>{azimuthal_angles,
                                                  elevation_angles,
                                                  rho_values}};

        // moving it back
        return std::move(spherical_vector);

    }
    // ============================================================================
    template <typename T>
    auto    sph2cart(const std::vector<T> spherical_vector){

        // creating cartesian vector
        auto    cartesian_vector  {std::vector<T>(spherical_vector.size(), 0)};

        // populating
        cartesian_vector[0]   =   spherical_vector[2] * \
                                  cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
                                  cos(spherical_vector[0] * std::numbers::pi / 180.00);
        cartesian_vector[1]   =   spherical_vector[2] * \
                                  cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
                                  sin(spherical_vector[0] * std::numbers::pi / 180.00);
        cartesian_vector[2]   =   spherical_vector[2] * \
                                  sin(spherical_vector[1] * std::numbers::pi / 180.00);

        // returning
        return std::move(cartesian_vector);
    }
}
```

## 10.8 Cosine

```
// ============================================================================
// y = cos(input_vector)
template <typename T>
auto cos(const std::vector<T>&  input_vector)
{
    // created canvas
    auto    canvas     {input_vector};

    // calling the function
    std::transform(input_vector.begin(), input_vector.end(),
                   canvas.begin(),
                   [](auto& argx){return std::cos(argx);});
```

```

    // returning the output
    return std::move(canvas);
}
// ============================================================================
// y = cosd(input_vector)
template <typename T>
auto cosd(std::vector<T>  input_vector)
{
    // created canvas
    auto    canvas     {input_vector};

    // calling the function
    std::transform(input_vector.begin(),
                   input_vector.end(),
                   input_vector.begin(),
                   [](const auto& argx){return std::cos(argx * 180.00/std::numbers::pi);});

    // returning the output
    return std::move(canvas);
}
```

## 10.9 Data Structures

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};


struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};
```

## 10.10 Editing Index Values

```
// ============================================================================
template <typename T, typename BooleanVector, typename U>
auto edit(std::vector<T>&               input_vector,
          BooleanVector                 bool_vector,
          const     U                   scalar)
{
    // throwing an error
    if (input_vector.size() != bool_vector.size())
        std::cerr << "edit: incompatible size\n";

    // overwriting input-vector
    std::transform(input_vector.begin(), input_vector.end(),
                   bool_vector.begin(),
                   input_vector.begin(),
                   [&scalar](auto& argx, auto argy){
                        if(argy == true)  {return static_cast<T>(scalar);}
                        else              {return argx;}
                   });

    // no-returns since in-place
}
```

## 10.11 Equality

```
// ============================================================================
template <typename T, typename U>
auto operator==(const std::vector<T>&   input_vector,
                const   U&               scalar)
{
    // setting up canvas
    auto    canvas     {std::vector<bool>(input_vector.size())};

    // writing to canvas
    std::transform(input_vector.begin(), input_vector.end(),
                   canvas.begin(),
                   [&scalar](const auto& argx){
                        return argx == scalar;
                   });

    // returning
    return std::move(canvas);
}
```

## 10.12 Exponentiate

```
// y = abs(vector)
template <typename T>
auto exp(const std::vector<T>&   input_vector)
{
    // creating canvas
    auto    canvas     {input_vector};

    // transforming
    std::transform(canvas.begin(), canvas.end(),
                   canvas.begin(),
                   [](auto& argx){return std::exp(argx);});

    // returning
    return std::move(canvas);
}
```

## 10.13 FFT

```
namespace  svr {
    // ============================================================================
    // For type-deductions
    template <typename T>
    struct fft_result_type;

    // specializations
    template <> struct fft_result_type<double>{
        using type = std::complex<double>;
    };
    template <> struct fft_result_type<std::complex<double>>{
        using type = std::complex<double>;
    };
    template <> struct fft_result_type<float>{
        using type = std::complex<float>;
    };
    template <> struct fft_result_type<std::complex<float>>{
        using type = std::complex<float>;
    };

    template <typename T>
    using   fft_result_t   = typename fft_result_type<T>::type;

    // // ========================================================================
```

```
// // y = fft(x, nfft)
// template<typename T>
// auto fft(const    std::vector<T>&   input_vector,
//         const     size_t            nfft)
// {

//    svr::Timer timer("fft");

//    // throwing an error
//    if (nfft < input_vector.size())  {std::cerr << "size-mistmatch\n";}
//    if (nfft <= 0)                   {std::cerr << "size-mistmatch\n";}

//    // fetching data-type
//    using  RType  = fft_result_t<T>;

//    // canvas instantiation
//    std::vector<RType> canvas(nfft);

//    // building time-only basis
//    std::vector<RType>
//    basiswithoutfrequency {linspace(static_cast<RType>(0),
//                                   static_cast<RType>(nfft-1),
//                                   nfft)};
//    auto   lambda_basiswithoutfrequency = [&basiswithoutfrequency](RType& arg){
//        return std::exp(-1.00 * 1i * 2.00 *                        \
//                        std::numbers::pi * static_cast<RType>(arg) / \
//                        static_cast<RType>(basiswithoutfrequency.size()));
//    };
//    std::transform(basiswithoutfrequency.begin(), basiswithoutfrequency.end(),
//                   basiswithoutfrequency.begin(),
//                   lambda_basiswithoutfrequency);

//    // building basis vectors
//    auto   bases_vectors  {std::vector<std::vector<RType>>()};
//    for(auto i = 0; i < nfft; ++i){
//        // making a copy of the bases-without-frequency
//        auto   temp   {basiswithoutfrequency};
//        // exponentiating basis with frequency
//        std::transform(temp.begin(), temp.end(),
//                       temp.begin(),
//                       [&i](auto& argx){return std::pow(argx, i);});
//        // pushing to end of bases-vectors
//        bases_vectors.push_back(std::move(temp));
//    }

//    // projecting input-array onto fourier bases
//    auto   finaloutput   {std::vector<RType>(nfft, 0)};
//    auto   nfft_sqrt     {static_cast<RType>(std::sqrt(nfft))};
//    #pragma omp parallel for
//    for(int i = 0; i < nfft; ++i){
//        // projecting input-vector with
//        finaloutput[i] =  std::inner_product(input_vector.begin(), input_vector.end(),
//                                             bases_vectors[i].begin(),
//                                             RType(0),
//                                             std::plus<RType>(),
//                                             [&nfft_sqrt](const auto& argx,
//                                                          const auto& argy){
//                                                 return static_cast<RType>(argx) *
//     static_cast<RType>(argy) / nfft_sqrt;
//                                             });
//    }
//    // returning finaloutput
//    return std::move(finaloutput);
// }

// =========================================================================
// y = fft(x, nfft)
template<typename T>
auto fft(const    std::vector<T>&   input_vector,
        const     size_t            nfft)
{
    // throwing an error
    if (nfft < input_vector.size())  {std::cerr << "size-mistmatch\n";}
    if (nfft <= 0)                   {std::cerr << "size-mistmatch\n";}

```

```
    // fetching data-type
    using  RType  = fft_result_t<T>;
    using  baseType   = std::conditional_t<std::is_same_v<T, std::complex<double>>,
                                          double,
                                          T>;

    // canvas instantiation
    std::vector<RType> canvas(nfft);
    auto   nfft_sqrt     {static_cast<RType>(std::sqrt(nfft))};
    auto   finaloutput   {std::vector<RType>(nfft, 0)};

    // calculating index by index
    for(int frequency_index = 0; frequency_index<nfft; ++frequency_index){
        RType  accumulate_value;
        for(int signal_index = 0; signal_index < input_vector.size(); ++signal_index){
            accumulate_value += \
                static_cast<RType>(input_vector[signal_index]) * \
                static_cast<RType>(std::exp(-1.00 * std::numbers::pi * \
                                   (static_cast<baseType>(frequency_index)/static_cast<baseType>(nfft))
                                        * \
                                   static_cast<baseType>(signal_index)));
        }
        finaloutput[frequency_index] = accumulate_value / nfft_sqrt;
    }

    // returning
    return std::move(finaloutput);
}











// // =========================================================================
// // y = ifft(x)
// template<typename T>
// auto ifft(const   std::vector<T>&   input_vector)
// {
//     svr::Timer timer00("ifft");

//     // fetching nfft
//     auto   nfft   {input_vector.size()};

//     // fetching data-type
//     using  RType  = fft_result_t<T>;

//     // canvas instantiation
//     std::vector<RType> canvas(nfft);

//     // building time-only basis
//     std::vector<RType>
//     basiswithoutfrequency {linspace(static_cast<RType>(0),
//                                     static_cast<RType>(nfft-1),
//                                     nfft)};
//     auto   lambda_basiswithoutfrequency = [&basiswithoutfrequency](RType& arg){
//         return std::exp(1.00 * 1i * 2.00 *                        \
//                         std::numbers::pi * static_cast<RType>(arg) / \
//                         static_cast<RType>(basiswithoutfrequency.size()));
//     };
//     std::transform(basiswithoutfrequency.begin(), basiswithoutfrequency.end(),
//                    basiswithoutfrequency.begin(),
//                    lambda_basiswithoutfrequency);

//     // building basis vectors
//     auto   bases_vectors  {std::vector<std::vector<RType>>()};
//     for(auto i = 0; i < nfft; ++i){
//         // making a copy of the bases-without-frequency
//         auto   temp   {basiswithoutfrequency};
```

```
    //        // exponentiating basis with frequency
    //        std::transform(temp.begin(), temp.end(),
    //                       temp.begin(),
    //                       [&i](auto& argx){return std::pow(argx, i);});
    //        // pushing to end of bases-vectors
    //        bases_vectors.push_back(std::move(temp));
    //    }

    //    // projecting input-array onto fourier bases
    //    auto    finaloutput    {std::vector<RType>(nfft, 0)};
    //    auto    nfft_sqrt      {static_cast<RType>(std::sqrt(nfft))};
    //    #pragma omp parallel for
    //    for(int i = 0; i < nfft; ++i){
    //        // projecting input-vector with
    //        finaloutput[i] =  std::inner_product(input_vector.begin(), input_vector.end(),
    //                                             bases_vectors[i].begin(),
    //                                             RType(0),
    //                                             std::plus<RType>(),
    //                                             [&nfft_sqrt](const auto& argx,
    //                                                          const auto& argy){
    //                                                  return static_cast<RType>(argx) * static_cast<RType>(argy) / nfft_sqrt;
    //                                              });
    //    }
    //    // returning finaloutput
    //    return std::move(finaloutput);
    // }

    // =====================================================================
    // y = ifft(x, nfft)
    template<typename T>
    auto ifft(const    std::vector<T>&   input_vector)
    {
        // fetching data-type
        using  RType   = fft_result_t<T>;
        using  baseType   = std::conditional_t<std::is_same_v<T, std::complex<double>>,
                                              double,
                                              T>;

        //  setup
        auto    nfft       {input_vector.size()};

        // canvas instantiation
        std::vector<RType> canvas(nfft);
        auto    nfft_sqrt      {static_cast<RType>(std::sqrt(nfft))};
        auto    finaloutput    {std::vector<RType>(nfft, 0)};

        // calculating index by index
        for(int frequency_index = 0; frequency_index<nfft; ++frequency_index){
            RType  accumulate_value;
            for(int signal_index = 0; signal_index < input_vector.size(); ++signal_index){
                accumulate_value += \
                    static_cast<RType>(input_vector[signal_index]) * \
                    static_cast<RType>(std::exp(1.00 * std::numbers::pi * \
                                             (static_cast<baseType>(frequency_index)/static_cast<baseType>(nfft)) * \
                                             static_cast<baseType>(signal_index)));
            }
            finaloutput[frequency_index] = accumulate_value / nfft_sqrt;
        }

        // returning
        return std::move(finaloutput);
    }
}
```

# 10.14 Flipping Containers

```
namespace svr {
    // =====================================================================
```

```
    template <typename T>
    auto   fliplr(const  std::vector<T>&   input_vector)
    {
        // creating canvas
        auto   canvas    {input_vector};

        // rewriting
        std::reverse(canvas.begin(), canvas.end());

        // returning
        return std::move(canvas);
    }
}
```

## 10.15 Indexing

```
namespace  svr {
    // ==========================================================================
    template   <typename T1, typename T2>
    auto   index(const   std::vector<T1>&     input_vector,
                 const   std::vector<T2>&     indices_to_sample)
    {
        // creating canvas
        auto   canvas    {std::vector<T1>(indices_to_sample.size(), 0)};

        // copying the associated values
        for(int i = 0; i < indices_to_sample.size(); ++i){
            auto   source_index   {indices_to_sample[i]};
            if(source_index < input_vector.size()){
                canvas[i] = input_vector[source_index];
            }
            else
                cout << "svr::index | source_index !< input_vector.size()\n";
        }

        // returning
        return std::move(canvas);
    }
}
```

## 10.16 Linspace

```
// in-place
template <typename T>
auto linspace(auto&        input,
              auto         startvalue,
              auto         endvalue,
              auto         numpoints) -> void
{
    auto stepsize = static_cast<T>(endvalue - startvalue)/static_cast<T>(numpoints-1);
    for(int i = 0; i<input.size(); ++i) {input[i] = startvalue + i*stepsize;}
};
// in-place
template <typename T>
auto linspace(vector<complex<T>>& input,
              auto                startvalue,
              auto                endvalue,
              auto                numpoints) -> void
{
    auto stepsize = static_cast<T>(endvalue - startvalue)/static_cast<T>(numpoints-1);
    for(int i = 0; i<input.size(); ++i) {
        input[i] = startvalue + static_cast<T>(i)*stepsize;
    }
};

// return-type
```

```
template <typename T>
auto linspace(T          startvalue,
             T          endvalue,
             size_t     numpoints)
{
    vector<T> input(numpoints);
    auto stepsize = static_cast<T>(endvalue - startvalue)/static_cast<T>(numpoints-1);

    for(int i = 0; i<input.size(); ++i) {input[i] = startvalue + static_cast<T>(i)*stepsize;}

    return input;
};

// return-type
template <typename T, typename U>
auto linspace(T          startvalue,
             U          endvalue,
             size_t     numpoints)
{
    vector<double> input(numpoints);
    auto stepsize = static_cast<double>(endvalue - startvalue)/static_cast<double>(numpoints-1);

    for(int i = 0; i<input.size(); ++i) {input[i] = startvalue + i*stepsize;}

    return input;
};
```

## 10.17 Max

```
template <std::size_t axis, typename T>
auto   max(const  std::vector<std::vector<T>>  input_matrix) -> std::enable_if_t<axis == 1,
     std::vector<std::vector<T>> >
{
    // setting up canvas
    auto   canvas    {std::vector<std::vector<T>>(input_matrix.size(),std::vector<T>(1))};

    // filling up the canvas
    for(auto row = 0; row < input_matrix.size(); ++row)
        canvas[row][0] = *(std::max_element(input_matrix[row].begin(), input_matrix[row].end()));

    // returning
    return std::move(canvas);
}
```

## 10.18 Meshgrid

```
// =============================================================================
template <typename T>
auto meshgrid(const   std::vector<T>&  x,
             const    std::vector<T>&  y)
{

    // creating and filling x-grid
    std::vector<std::vector<T>> xcanvas(y.size(), std::vector<T>(x.size(), 0));
    for(auto row = 0; row < y.size(); ++row)
        std::copy(x.begin(), x.end(), xcanvas[row].begin());

    // creating and filling y-grid
    std::vector<std::vector<T>> ycanvas(y.size(), std::vector<T>(x.size(), 0));
    for(auto col = 0; col < x.size(); ++col)
        for(auto row = 0; row < y.size(); ++row)
            ycanvas[row][col] = y[row];

    // returning
    return std::move(std::pair{xcanvas, ycanvas});

```

```
}
// ========================================================================
template <typename T>
auto meshgrid(std::vector<T>&& x,
              std::vector<T>&&  y)
{

    // creating and filling x-grid
    std::vector<std::vector<T>> xcanvas(y.size(), std::vector<T>(x.size(), 0));
    for(auto row = 0; row < y.size(); ++row)
        std::copy(x.begin(), x.end(), xcanvas[row].begin());

    // creating and filling y-grid
    std::vector<std::vector<T>> ycanvas(y.size(), std::vector<T>(x.size(), 0));
    for(auto col = 0; col < x.size(); ++col)
        for(auto row = 0; row < y.size(); ++row)
            ycanvas[row][col] = y[row];

    // returning
    return std::move(std::pair{xcanvas, ycanvas});

}
```

## 10.19 Minimum

```
template <std::size_t axis, typename T>
auto    min(std::vector<std::vector<T>> input_matrix) -> std::enable_if_t<axis == 1,
     std::vector<std::vector<T>> >
{
    // creating canvas
    auto    canvas     {std::vector<std::vector<T>>(input_matrix.size(),std::vector<T>(1))};

    // storing the values
    for(auto row = 0; row < input_matrix.size(); ++row)
        canvas[row][0]   = *(std::min_element(input_matrix[row].begin(), input_matrix[row].end()));

    // returning the value
    return std::move(canvas);
}
```

## 10.20 Norm

```
// ========================================================
template <typename T>
auto norm(const   std::vector<T>&   input_vector)
{
    return std::sqrt(std::inner_product(input_vector.begin(), input_vector.end(),
                                        input_vector.begin(),
                                        (T)0));
}



/*
Templates to create
    -   matrix and norm-axis
    -   axis instantiated std::vector<T>
*/
```

## 10.21 Division

```
// =========================================================
// matrix division with scalars
template <typename T>
auto operator/(const  std::vector<T>&   input_vector,
               const    T&                input_scalar)
{
    // creating canvas
    auto    canvas     {input_vector};

    // filling canvas
    std::transform(canvas.begin(), canvas.end(),
                   canvas.begin(),
                   [&input_scalar](const auto& argx){
                        return static_cast<double>(argx) / static_cast<double>(input_scalar);
                   });

    // returning value
    return std::move(canvas);
}
// =========================================================
// matrix division with scalars
template <typename T>
auto operator/=(const  std::vector<T>&   input_vector,
                const    T&                input_scalar)
{
    // creating canvas
    auto    canvas     {input_vector};

    // filling canvas
    std::transform(canvas.begin(), canvas.end(),
                   canvas.begin(),
                   [&input_scalar](const auto& argx){
                        return static_cast<double>(argx) / static_cast<double>(input_scalar);
                   });

    // returning value
    return std::move(canvas);
}
```

## 10.22 Addition

```
// ===========================================================================
// y = vector + vector
template <typename T>
std::vector<T> operator+(const std::vector<T>& a,
                         const std::vector<T>& b)
{
    // Identify which is bigger
    const auto& big = (a.size() > b.size()) ? a : b;
    const auto& small = (a.size() > b.size()) ? b : a;

    std::vector<T> result = big; // copy the bigger one

    // Add elements from the smaller one
    for (size_t i = 0; i < small.size(); ++i) {
        result[i] += small[i];
    }

    return result;
}
// ===========================================================================
// y = vector + vector
template <typename T>
std::vector<T>& operator+=(std::vector<T>& a,
                           const std::vector<T>& b) {

    const auto& small = (a.size() < b.size()) ? a : b;
    const auto& big = (a.size() < b.size()) ? b : a;

    // If b is bigger, resize 'a' to match
```

```
    if (a.size() < b.size())                   {a.resize(b.size());}

    // Add elements
    for (size_t i = 0; i < small.size(); ++i)  {a[i] += b[i];}

    // returning elements
    return a;
}
// =============================================================================
// y = matrix + matrix
template <typename T>
std::vector<std::vector<T>> operator+(const std::vector<std::vector<T>>& a,
                                      const std::vector<std::vector<T>>& b)
{
    // fetching dimensions
    const  auto&  num_rows_A    {a.size()};
    const  auto&  num_cols_A    {a[0].size()};
    const  auto&  num_rows_B    {b.size()};
    const  auto&  num_cols_B    {b[0].size()};

    // choosing the three different metrics
    if (num_rows_A != num_rows_B && num_cols_A != num_cols_B){
        cout << format("a.dimensions = [{},{}], b.shape = [{},{}]\n",
                       num_rows_A, num_cols_A,
                       num_rows_B, num_cols_B);
        std::cerr << "dimensions don't match\n";
    }

    // creating canvas
    auto   canvas     {std::vector<std::vector<T>>(
        std::max(num_rows_A, num_rows_B),
        std::vector<T>(std::max(num_cols_A, num_cols_B), (T)0.00)
    )};

    // performing addition
    if (num_rows_A == num_rows_B && num_cols_A == num_cols_B){
        for(auto row = 0; row < num_rows_A; ++row){
            std::transform(a[row].begin(), a[row].end(),
                           b[row].begin(),
                           canvas[row].begin(),
                           std::plus<T>());
        }
    }
    else if(num_rows_A == num_rows_B){

        // if number of columsn are different, check if one of the cols are one
        const  auto   min_num_cols   {std::min(num_cols_A, num_cols_B)};
        if (min_num_cols != 1) {std::cerr<< "Operator+: unable to broadcast\n";}
        const  auto   max_num_cols   {std::max(num_cols_A, num_cols_B)};

        // using references to tag em differently
        const  auto&  big_matrix     {num_cols_A > num_cols_B ? a : b};
        const  auto&  small_matrix   {num_cols_A < num_cols_B ? a : b};

        // Adding to canvas
        for(auto row = 0; row < canvas.size(); ++row){
            std::transform(big_matrix[row].begin(), big_matrix[row].end(),
                           canvas[row].begin(),
                           [&small_matrix,
                            &row](const auto& argx){
                               return argx + small_matrix[row][0];
                           });
        }
    }
    else if(num_cols_A == num_cols_B){

        // check if the smallest column-number is one
        const  auto   min_num_rows   {std::min(num_rows_A, num_rows_B)};
        if(min_num_rows != 1)     {std::cerr << "Operator+ : unable to broadcast\n";}
        const  auto   max_num_rows   {std::max(num_rows_A, num_rows_B)};

        // using references to differentiate the two matrices
        const  auto&  big_matrix     {num_rows_A > num_rows_B ? a : b};
        const  auto&  small_matrix   {num_rows_A < num_rows_B ? a : b};

```

```
        // adding to canvas
        for(auto row = 0; row < canvas.size(); ++row){
            std::transform(big_matrix[row].begin(), big_matrix[row].end(),
                           small_matrix[0].begin(),
                           canvas[row].begin(),
                           [](const auto& argx, const auto& argy){
                            return argx + argy;
                           });
        }
    }
    else {
        PRINTLINE PRINTLINE PRINTLINE PRINTLINE PRINTLINE
        cout << format("check this again \n");
    }

    // returning
    return std::move(canvas);
}
// ===========================================================================
// y = vector + scalar
template <typename T>
auto operator+(const  std::vector<T>&   input_vector,
               const   T                 scalar)
{
    // creating canvas
    auto    canvas     {input_vector};

    // adding scalar to the canvas
    std::transform(canvas.begin(), canvas.end(),
                   canvas.begin(),
                   [&scalar](auto& argx){return argx + scalar;});

    // returning canvas
    return std::move(canvas);
}
// ===========================================================================
// y = scalar + vector
template <typename T>
auto operator+(const  T                  scalar,
               const   std::vector<T>&   input_vector)
{
    // creating canvas
    auto    canvas     {input_vector};

    // adding scalar to the canvas
    std::transform(canvas.begin(), canvas.end(),
                   canvas.begin(),
                   [&scalar](auto& argx){return argx + scalar;});

    // returning canvas
    return std::move(canvas);
}
```

## 10.23 Multiplication (Element-wise)

```
// scalar * vector ============================================================
template <typename T>
auto    operator*(const   T                  scalar,
                  const    std::vector<T>&   input_vector)
{
    // creating canvas
    auto    canvas     {input_vector};
    // performing operation
    std::for_each(canvas.begin(), canvas.end(),
                  [&scalar](auto& argx){argx = argx * scalar;});
    // returning
    return std::move(canvas);
}

// scalar * vector ============================================================
```

```
// template <typename T1, typename T2>
template <typename T1, typename T2,
          typename = std::enable_if_t<!std::is_same_v<std::decay_t<T1>, std::vector<T2>>>>
auto operator*(const  T1                scalar,
               const   vector<T2>&       input_vector)
{
    // fetching final-type
    using T3 = decltype(std::declval<T1>() * std::declval<T2>());
    // creating canvas
    auto    canvas     {std::vector<T3>(input_vector.size())};
    // multiplying
    std::transform(input_vector.begin(), input_vector.end(),
                   canvas.begin(),
                   [&scalar](auto& argx){
                    return static_cast<T3>(scalar) * static_cast<T3>(argx);
                   });
    // returning
    return std::move(canvas);
}

// vector * scalar ============================================================
template <typename T>
auto    operator*(const    std::vector<T>&   input_vector,
                  const     T                  scalar)
{
    // creating canvas
    auto    canvas     {input_vector};
    // multiplying
    std::for_each(canvas.begin(), canvas.end(),
                  [&scalar](auto& argx){
                    argx = argx * scalar;
                  });
    // returning
    return std::move(canvas);
}

// vector * vector ============================================================
template <typename T>
auto operator*(const std::vector<T>& input_vector_A,
               const std::vector<T>&  input_vector_B)
{
    // throwing error: size-desparity
    if (input_vector_A.size() != input_vector_B.size()) {std::cerr << "operator*: size disparity \n";}

    // creating canvas
    auto    canvas     {input_vector_A};

    // element-wise multiplying
    std::transform(input_vector_B.begin(), input_vector_B.end(),
                   canvas.begin(),
                   canvas.begin(),
                   [](const auto& argx, const auto& argy){
                       return argx * argy;
                   });

    // moving it back
    return std::move(canvas);
}
template <typename T1, typename T2>
auto    operator*(const    std::vector<T1>&  input_vector_A,
                  const     std::vector<T2>&  input_vector_B)
{

    // checking size disparity
    if (input_vector_A.size() != input_vector_B.size())
        std::cerr << "operator*: error, size-disparity \n";

    // figuring out resulting data type
    using   T3    = decltype(std::declval<T1>() * std::declval<T2>());

    // creating canvas
    auto    canvas     {std::vector<T3>(input_vector_A.size())};

    // performing multiplications
    std::transform(input_vector_A.begin(), input_vector_A.end(),
```

```
                 input_vector_B.begin(),
                 canvas.begin(),
                 [](const    auto&      argx,
                    const    auto&      argy){
                      return static_cast<T3>(argx) *  static_cast<T3>(argy);
                 });

    // returning
    return std::move(canvas);

}

// scalar * matrix ============================================================
template <typename T>
auto operator*(T                               scalar,
               const std::vector<std::vector<T>>& inputMatrix)
{
    std::vector<std::vector<T>> temp {inputMatrix};
    for(int i = 0; i<inputMatrix.size(); ++i){
        std::transform(inputMatrix[i].begin(),
                       inputMatrix[i].end(),
                       temp[i].begin(),
                       [&scalar](T x){return scalar * x;});
    }
    return temp;
}
// matrix * matrix ============================================================
template <typename T>
auto operator*(const std::vector<std::vector<T>>& A,
               const std::vector<std::vector<T>>& B) -> std::vector<std::vector<T>>
{
    // Case 1: element-wise multiplication
    if (A.size() == B.size() && A[0].size() == B[0].size()) {
        std::vector<std::vector<T>> C(A.size(), std::vector<T>(A[0].size()));
        for (std::size_t row = 0; row < A.size(); ++row) {
            std::transform(A[row].begin(), A[row].end(),
                           B[row].begin(),
                           C[row].begin(),
                           [](const auto& x, const auto& y){ return x * y; });
        }
        return C;
    }

    // Case 2: broadcast column vector
    else if (A.size() == B.size() && B[0].size() == 1) {
        std::vector<std::vector<T>> C(A.size(), std::vector<T>(A[0].size()));
        for (std::size_t row = 0; row < A.size(); ++row) {
            std::transform(A[row].begin(), A[row].end(),
                           C[row].begin(),
                           [&](const auto& x){ return x * B[row][0]; });
        }
        return C;
    }

    // case 3: when second matrix contains just one row
    // case 4: when first matrix is just one column
    // case 5: when second matrix is just one column

    // Otherwise, invalid
    else {
        throw std::runtime_error("operator* dimension mismatch");
    }
}
// scalar * matrix ============================================================
template <typename T1, typename T2>
auto operator*(T1 scalar,
               const std::vector<std::vector<T2>>& inputMatrix)
{
    std::vector<std::vector<T2>> temp {inputMatrix};
    for(int i = 0; i<inputMatrix.size(); ++i){
        std::transform(inputMatrix[i].begin(),
                       inputMatrix[i].end(),
                       temp[i].begin(),
                       [&scalar](T2 x){return static_cast<T2>(scalar) * x;});
    }
```

```
    return temp;
}
// // scalar * matrix ==============================================================
// template <typename T1,
//          typename T2,
//          typename = typename std::enable_if<std::is_arithmetic<T1>::value>::type>
// auto operator*(T1 scalar,
//               const std::vector<std::vector<T2>>& inputMatrix)
// {
//     std::vector<std::vector<T2>> temp {inputMatrix};
//     for(int i = 0; i<inputMatrix.size(); ++i){
//         std::transform(inputMatrix[i].begin(),
//                        inputMatrix[i].end(),
//                        temp[i].begin(),
//                        [&scalar](T2 x){return static_cast<T2>(scalar) * x;});
//     }
//     return temp;
// }
// matrix-multiplication =======================================================
template <typename T1, typename T2>
auto matmul(const std::vector<std::vector<T1>>& matA,
            const std::vector<std::vector<T2>>& matB)
{

    // throwing error
    if (matA[0].size() != matB.size())  {std::cerr << "dimension-mismatch \n";}

    // getting result-type
    using ResultType  = decltype(std::declval<T1>() * std::declval<T2>() + \
                                std::declval<T1>() * std::declval<T2>() );

    // creating aliasses
    auto finalnumrows {matA.size()};
    auto finalnumcols {matB[0].size()};

    // creating placeholder
    auto rowcolproduct = [&](auto rowA, auto colB){
        ResultType temp {0};
        for(int i = 0; i < matA.size(); ++i) {temp += static_cast<ResultType>(matA[rowA][i]) +
             static_cast<ResultType>(matB[i][colB]);}
        return temp;
    };

    // producing row-column combinations
    std::vector<std::vector<ResultType>> finaloutput(finalnumrows, std::vector<ResultType>(finalnumcols));
    for(int row = 0; row < finalnumrows; ++row){for(int col = 0; col < finalnumcols;
         ++col){finaloutput[row][col] = rowcolproduct(row, col);}}

    // returning
    return finaloutput;
}
// matrix * vector ==============================================================
template   <typename T>
auto operator*(const  std::vector<std::vector<T>>  input_matrix,
               const   std::vector<T>              input_vector)
{
    // fetching dimensions
    const  auto&  num_rows_matrix   {input_matrix.size()};
    const  auto&  num_cols_matrix   {input_matrix[0].size()};
    const  auto&  num_rows_vector   {1};
    const  auto&  num_cols_vector   {input_vector.size()};

    const  auto&  max_num_rows      {num_rows_matrix > num_rows_vector ?\
                                     num_rows_matrix : num_rows_vector};
    const  auto&  max_num_cols      {num_cols_matrix > num_cols_vector ?\
                                     num_cols_matrix : num_cols_vector};

    // creating canvas
    auto    canvas     {std::vector<std::vector<T>>(
        max_num_rows,
        std::vector<T>(max_num_cols, 0)
    )};

    //
    if (num_cols_matrix == 1 && num_rows_vector == 1){
```

```cpp

        // writing to canvas
        for(auto   row = 0; row < max_num_rows; ++row)
            for(auto   col = 0; col < max_num_cols; ++col)
                canvas[row][col]  =  input_matrix[row][0]  *  input_vector[col];
    }
    else{
        std::cerr << "Operator*: [matrix, vector] | not implemented \n";
    }

    // returning
    return std::move(canvas);

}

// scalar operators ============================================================
auto operator*(const std::complex<double> complexscalar,
               const double               doublescalar){
    return complexscalar * static_cast<std::complex<double>>(doublescalar);
}
auto operator*(const double               doublescalar,
               const std::complex<double> complexscalar){
    return complexscalar * static_cast<std::complex<double>>(doublescalar);
}
auto operator*(const std::complex<double> complexscalar,
               const int                  scalar){
    return complexscalar * static_cast<std::complex<double>>(scalar);
}
auto operator*(const int                  scalar,
               const std::complex<double> complexscalar){
    return complexscalar * static_cast<std::complex<double>>(scalar);
}
```

## 10.24 Subtraction

```cpp
// =============================================================================
// Aim: substracting scalar from a vector
template <typename T>
std::vector<T> operator-(const std::vector<T>& a, const T scalar){
    std::vector<T> temp(a.size());
    std::transform(a.begin(),
                   a.end(),
                   temp.begin(),
                   [scalar](T x){return (x - scalar);});
    return temp;
}
// =============================================================================
template <typename T>
auto operator-(const  std::vector<std::vector<T>>&  input_matrix_A,
               const   std::vector<std::vector<T>>&  input_matrix_B)
{
    // throwing error in case of dimension differences
    if (input_matrix_A.size() != input_matrix_B.size() ||
        input_matrix_A[0].size() != input_matrix_B[0].size())
        std::cerr << "operator- dimension mismatch\n";

    // setting up canvas
    auto    canvas     {std::vector<std::vector<T>>(
        input_matrix_A.size(),
        std::vector<T>(input_matrix_A[0].size())
    )};

    // subtracting values
    for(auto row = 0; row < input_matrix_B.size(); ++row)
        std::transform(input_matrix_A[row].begin(), input_matrix_A[row].end(),
                       input_matrix_B[row].begin(),
                       canvas[row].begin(),
                       [](const auto& x, const auto& y){
                           return x - y;
                       });
```

```

    // returning
    return std::move(canvas);

}
```

## 10.25 Operator Overloadings

## 10.26 Printing Containers

```
// vector printing function
template<typename T>
void fPrintVector(vector<T> input){
    for(auto x: input) cout << x << ",";
    cout << endl;
}

template<typename T>
void fpv(vector<T> input){
    for(auto x: input) cout << x << ",";
    cout << endl;
}
// =========================================================
template<typename T>
void fPrintMatrix(const std::vector<std::vector<T>> input_matrix){
    for(const auto& row: input_matrix)
        cout << format("{}\n", row);
}
template <typename T>
void fPrintMatrix(const string&                input_string,
                  const std::vector<std::vector<T>> input_matrix){
    cout << format("{} = \n", input_string);
    for(const auto& row: input_matrix)
        cout << format("{}\n", row);
}


template<typename T, typename T1>
void fPrintHashmap(unordered_map<T, T1> input){
    for(auto x: input){
        cout << format("[{},{}] | ", x.first, x.second);
    }
    cout <<endl;
}

void fPrintBinaryTree(TreeNode* root){
    // sending it back
    if (root == nullptr) return;

    // printing
    PRINTLINE
    cout << "root->val = " << root->val << endl;

    // calling the children
    fPrintBinaryTree(root->left);
    fPrintBinaryTree(root->right);

    // returning
    return;

}

void fPrintLinkedList(ListNode* root){
    if (root == nullptr) return;
    cout << root->val << " -> ";
```

```
    fPrintLinkedList(root->next);
    return;
}

template<typename T>
void fPrintContainer(T input){
    for(auto x: input) cout << x << ", ";
    cout << endl;
    return;
}
// =============================================================================
template <typename T>
auto size(std::vector<std::vector<T>> inputMatrix){
    cout << format("[{}, {}]\n", inputMatrix.size(), inputMatrix[0].size());
}

template <typename T>
auto size(const std::string inputstring, std::vector<std::vector<T>> inputMatrix){
    cout << format("{} = [{}, {}]\n", inputstring, inputMatrix.size(), inputMatrix[0].size());
}
```

## 10.27 Random Number Generation

```
// =============================================================================
template <typename T>
auto rand(const T min, const T max) {
    static std::random_device rd; // Seed
    static std::mt19937 gen(rd()); // Mersenne Twister generator
    std::uniform_real_distribution<> dist(min, max);
    return dist(gen);
}
// =============================================================================
template <typename T>
auto rand(const T     min,
          const T     max,
          const size_t numelements)
{
    static std::random_device rd; // Seed
    static std::mt19937 gen(rd()); // Mersenne Twister generator
    std::uniform_real_distribution<> dist(min, max);

    // building the fianloutput
    vector<T> finaloutput(numelements);
    for(int i = 0; i<finaloutput.size(); ++i) {finaloutput[i] = static_cast<T>(dist(gen));}

    return finaloutput;
}
// =============================================================================
template <typename T>
auto rand(const T            argmin,
          const T            argmax,
          const vector<int>  dimensions)
{

    // throwing an error if dimension is greater than two
    if (dimensions.size() > 2) {std::cerr << "dimensions are too high\n";}

    // creating random engine
    static std::random_device rd; // Seed
    static std::mt19937 gen(rd()); // Mersenne Twister generator
    std::uniform_real_distribution<> dist(argmin, argmax);

    // building the finaloutput
    vector<vector<T>> finaloutput;
    for(int i = 0; i<dimensions[0]; ++i){
        vector<T> temp;
        for(int j = 0; j<dimensions[1]; ++j) {temp.push_back(dist(gen));}
        // cout << format("\t\t temp = {}\n", temp);

        finaloutput.push_back(temp);
```

```
    }

    // returning the finaloutput
    return finaloutput;

}
// =============================================================================
auto rand(const vector<int>  dimensions)
{

    using ReturnType = double;

    // throwing an error if dimension is greater than two
    if (dimensions.size() > 2) {std::cerr << "dimensions are too high\n";}

    // creating random engine
    static std::random_device rd; // Seed
    static std::mt19937 gen(rd()); // Mersenne Twister generator
    std::uniform_real_distribution<> dist(0.00, 1.00);

    // building the finaloutput
    vector<vector<ReturnType>> finaloutput;
    for(int i = 0; i<dimensions[0]; ++i){
        vector<ReturnType> temp;
        for(int j = 0; j<dimensions[1]; ++j) {temp.push_back(dist(gen));}
        finaloutput.push_back(std::move(temp));
    }

    // returning the finaloutput
    return std::move(finaloutput);

}
// =============================================================================
template <typename T>
auto rand_complex_double(const T           argmin,
                         const T           argmax,
                         const vector<int>& dimensions)
{

    // throwing an error if dimension is greater than two
    if (dimensions.size() > 2) {std::cerr << "dimensions are too high\n";}

    // creating random engine
    static std::random_device rd; // Seed
    static std::mt19937 gen(rd()); // Mersenne Twister generator
    std::uniform_real_distribution<> dist(argmin, argmax);

    // building the finaloutput
    vector<vector<complex<double>>> finaloutput;
    for(int i = 0; i<dimensions[0]; ++i){
        vector<complex<double>> temp;
        for(int j = 0; j<dimensions[1]; ++j) {temp.push_back(static_cast<double>(dist(gen)));}
        finaloutput.push_back(std::move(temp));
    }

    // returning the finaloutput
    return finaloutput;

}
```

## 10.28 Reshape

```
// ========================================================================
// reshaping a matrix into another matrix
template <std::size_t M, std::size_t N, typename T>
auto reshape(const std::vector<std::vector<T>>& input_matrix){

    // verifying size stuff
    if (M*N != input_matrix.size() * input_matrix[0].size())
        std::cerr << "Dimensions are quite different\n";
```

```

    // creating canvas
    auto   canvas     {std::vector<std::vector<T>>(
        M, std::vector<T>(N, (T)0)
    )};

    // writing to canvas
    size_t   tid           {0};
    size_t   target_row    {0};
    size_t   target_col    {0};
    for(auto row = 0; row<input_matrix.size(); ++row){
        for(auto col = 0; col < input_matrix[0].size(); ++col){
            tid           =   row * input_matrix[0].size() + col;
            target_row    =   tid/N;
            target_col    =   tid%N;
            canvas[target_row][target_col]   =   input_matrix[row][col];
        }
    }

    // moving it back
    return std::move(canvas);
}
// =========================================================================
// reshaping a matrix into a vector
template<std::size_t M, typename T>
auto reshape(const std::vector<std::vector<T>>& input_matrix){

    // checking element-count validity
    if (M != input_matrix.size() * input_matrix[0].size())
        std::cerr << "Number of elements differ\n";

    // creating canvas
    auto   canvas     {std::vector<T>(M, 0)};

    // filling canvas
    for(auto row = 0; row < input_matrix.size(); ++row)
        for(auto col = 0; col < input_matrix[0].size(); ++col)
            canvas[row * input_matrix.size() + col] = input_matrix[row][col];

    // moving it back
    return std::move(canvas);
}

// ==========================================================
// Matrix to matrix
// ==========================================================
template<typename T>
auto reshape(const std::vector<std::vector<T>>& input_matrix,
             const std::size_t               M,
             const std::size_t               N){

    // checking element-count validity
    if ( M * N != input_matrix.size() * input_matrix[0].size())
        std::cerr << "Number of elements differ\n";

    // creating canvas
    auto   canvas     {std::vector<std::vector<T>>(
        M, std::vector<T>(N, (T)0)
    )};

    // writing to canvas
    size_t   tid           {0};
    size_t   target_row    {0};
    size_t   target_col    {0};
    for(auto row = 0; row<input_matrix.size(); ++row){
        for(auto col = 0; col < input_matrix[0].size(); ++col){
            tid           =   row * input_matrix[0].size() + col;
            target_row    =   tid/N;
            target_col    =   tid%N;
            canvas[target_row][target_col]   =   input_matrix[row][col];
        }
    }

    // moving it back
    return std::move(canvas);
```

```
}

// ========================================================
// converting a matrix into a vector
// ========================================================
template<typename T>
auto reshape(const std::vector<std::vector<T>>& input_matrix,
             const size_t                    M){

    // checking element-count validity
    if (M != input_matrix.size() * input_matrix[0].size())
        std::cerr << "Number of elements differ\n";

    // creating canvas
    auto    canvas     {std::vector<T>(M, 0)};

    // filling canvas
    for(auto row = 0; row < input_matrix.size(); ++row)
        for(auto col = 0; col < input_matrix[0].size(); ++col)
            canvas[row * input_matrix.size() + col] = input_matrix[row][col];

    // moving it back
    return std::move(canvas);
}
```

## 10.29 Summing with containers

```
// ============================================================================
template <std::size_t axis, typename T>
auto sum(const std::vector<T>&  input_vector) -> std::enable_if_t<axis == 0, std::vector<T>>
{
    // returning the input as is
    return input_vector;
}
// ============================================================================
template <std::size_t axis, typename T>
auto sum(const std::vector<std::vector<T>>& input_matrix) -> std::enable_if_t<axis == 0, std::vector<T>>
{
    // creating canvas
    auto    canvas     {std::vector<T>(input_matrix[0].size(), 0)};

    // filling up the canvas
    for(auto row = 0; row < input_matrix.size(); ++row)
        std::transform(input_matrix[row].begin(), input_matrix[row].end(),
                       canvas.begin(),
                       canvas.begin(),
                       [](auto& argx, auto& argy){return argx + argy;});

    // returning
    return std::move(canvas);

}
// ============================================================================
template <std::size_t axis, typename T>
auto sum(const std::vector<std::vector<T>>& input_matrix) -> std::enable_if_t<axis == 1,
      std::vector<std::vector<T>>>
{
    // creating canvas
    auto canvas    {std::vector<std::vector<T>>(input_matrix.size(),
                                                std::vector<T>(1, 0.00))};

    // filling up the canvas
    for(auto row = 0; row < input_matrix.size(); ++row)
        canvas[row][0]    = std::accumulate(input_matrix[row].begin(),
                                            input_matrix[row].end(),
                                            static_cast<T>(0));

    // returning
    return std::move(canvas);

```

```
}
// ============================================================================
template <std::size_t axis, typename T>
auto sum(const std::vector<T>&  input_vector_A,
        const std::vector<T>&   input_vector_B) -> std::enable_if_t<axis == 0, std::vector<T> >
{
    // setup
    const   auto&   num_cols_A      {input_vector_A.size()};
    const   auto&   num_cols_B      {input_vector_B.size()};

    // throwing errors
    if (num_cols_A != num_cols_B) {std::cerr << "sum: size disparity\n";}

    // creating canvas
    auto    canvas {input_vector_A};

    // summing up
    std::transform(input_vector_B.begin(), input_vector_B.end(),
                   canvas.begin(),
                   canvas.begin(),
                   std::plus<T>());

    // returning
    return std::move(canvas);
}
```

## 10.30 Tangent

```
namespace  svr {
    // ====================================================
    template <typename T>
    auto atan2(const  std::vector<T>    input_vector_A,
               const    std::vector<T>    input_vector_B)
    {
        // throw error
        if (input_vector_A.size() != input_vector_B.size())
            std::cerr << "atan2: size disparity\n";

        // create canvas
        auto    canvas     {std::vector<T>(input_vector_A.size(), 0)};

        // performing element-wise atan2 calculation
        std::transform(input_vector_A.begin(), input_vector_A.end(),
                       input_vector_B.begin(),
                       canvas.begin(),
                       [](const    auto&  arg_a,
                          const    auto&  arg_b){

                            return std::atan2(arg_a, arg_b);
                       });

        // moving things back
        return std::move(canvas);
    }
    // ====================================================
    template <typename T>
    auto atan2(T   scalar_A,
               T   scalar_B)
    {
        return std::atan2(scalar_A, scalar_B);
    }
}
```

## 10.31 Tiling Operations

```
namespace  svr {
```

```
    // ======================================================
    template <typename T>
    auto tile(const    std::vector<T>&      input_vector,
              const    std::vector<size_t>  mul_dimensions){

        // creating canvas
        const  std::size_t& num_rows   {1 * mul_dimensions[0]};
        const  std::size_t& num_cols   {input_vector.size() * mul_dimensions[1]};
        auto   canvas {std::vector<std::vector<T>>(
            num_rows,
            std::vector<T>(num_cols, 0)
        )};

        // writing
        std::size_t   source_row;
        std::size_t   source_col;

        for(std::size_t row = 0; row < num_rows; ++row){
            for(std::size_t col = 0; col < num_cols; ++col){
                source_row =  row % 1;
                source_col =  col % input_vector.size();
                canvas[row][col] = input_vector[source_col];
            }
        }

        // returning
        return std::move(canvas);
    }
    // ======================================================
    template <typename T>
    auto tile(const    std::vector<std::vector<T>>& input_matrix,
              const    std::vector<size_t>          mul_dimensions){

        // creating canvas
        const  std::size_t& num_rows   {input_matrix.size()   * mul_dimensions[0]};
        const  std::size_t& num_cols   {input_matrix[0].size() * mul_dimensions[1]};
        auto   canvas {std::vector<std::vector<T>>(
            num_rows,
            std::vector<T>(num_cols, 0)
        )};

        // writing
        std::size_t   source_row;
        std::size_t   source_col;

        for(std::size_t row = 0; row < num_rows; ++row){
            for(std::size_t col = 0; col < num_cols; ++col){
                source_row =  row % input_matrix.size();
                source_col =  col % input_matrix[0].size();
                canvas[row][col] = input_matrix[source_row][source_col];
            }
        }

        // returning
        return std::move(canvas);
    }
}
```

## 10.32 Transpose

```
template <typename T>
auto transpose(const std::vector<T> input_vector){

    // creating canvas
    auto    canvas     {std::vector<std::vector<T>>{
        input_vector.size(),
        std::vector<T>(1)
    }};

    // filling canvas
```

```
    for(auto i = 0; i < input_vector.size(); ++i){
        canvas[i][0]   = input_vector[i];
    }

    // moving it back
    return std::move(canvas);
}
```