

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

Preface	i
Introduction	ii
1 Setup	1
1.1 Overview	1
2 Underwater Environment Setup	2
2.1 Sea “Floor”	2
2.2 Simple Structures	3
2.2.1 Boxes	3
2.2.2 Sphere	4
3 Hardware Setup	5
3.1 Transmitter	5
3.2 Uniform Linear Array	6
3.3 Marine Vessel	6
4 Signal Simulation	7
4.1 Transmitted Signal	7
4.2 Signal Simulation	8
4.3 Ray Tracing	8
4.3.1 Pairwise Dot-Product	8
4.3.2 Range Histogram Method	9
5 Imaging	10
5.1 Decimation	10
5.1.1 Basebanding	10
5.1.2 Lowpass filtering	11
5.1.3 Decimation	11
5.2 Match-Filtering	11

6	Control Pipeline	14
7	Results	16
8	Software	17
8.1	Class Definitions	17
8.1.1	Class: Scatter	17
8.1.2	Class: Transmitter	20
8.1.3	Class: Uniform Linear Array	28
8.1.4	Class: Autonomous Underwater Vehicle	47
8.2	Setup Scripts	57
8.2.1	Seafloor Setup	57
8.2.2	Transmitter Setup	59
8.2.3	ULA Setup	61
8.2.4	AUV Setup	63
8.3	Function Definitions	64
8.3.1	Cartesian Coordinates to Spherical Coordinates	64
8.3.2	Spherical Coordinates to Cartesian Coordinates	65
8.3.3	Column-Wise Convolution	65
8.3.4	Buffer 2D	66
8.3.5	fAnglesToTensor	67
8.3.6	fCalculateCosine	68
8.4	Main Scripts	69
8.4.1	Signal Simulation	69
9	Reading	72
9.1	Primary Books	72
9.2	Interesting Papers	72
10	General Purpose Templated Functions	73
10.1	CSV File-Writes	73
10.2	abs	74
10.3	Boolean Comparators	75
10.4	Concatenate Functions	76
10.5	Conjugate	77
10.6	Convolution	77
10.7	Coordinate Change	78
10.8	Cosine	78

10.9	Data Structures	79
10.10	Editing Index Values	79
10.11	Equality	80
10.12	Exponentiate	80
10.13	FFT	80
10.14	Flipping Containers	83
10.15	Indexing	84
10.16	Linspace	84
10.17	Max	85
10.18	Meshgrid	85
10.19	Minimum	86
10.20	Norm	86
10.21	Division	86
10.22	Addition	87
10.23	Multiplication (Element-wise)	89
10.24	Subtraction	93
10.25	Operator Overloadings	94
10.26	Printing Containers	94
10.27	Random Number Generation	95
10.28	Reshape	96
10.29	Summing with containers	98
10.30	Tangent	99
10.31	Tiling Operations	99
10.32	Transpose	100

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

```

1: Input: Mean vector  $\mathbf{m}$ , Dimension vector  $\mathbf{d}$ , 2D points  $\mathbf{P}$ 
2: Output: Updated  $\mathbf{P}$  with hill heights
3:  $\text{num\_hills} \leftarrow \text{numel}(\mathbf{m}_x)$ 
4:  $H \leftarrow$  Zeros tensor of size  $(1, \text{numel}(\mathbf{P}_x))$ 
5: for  $i = 1$  to  $\text{num\_hills}$  do
6:    $x_{\text{norm}} \leftarrow \frac{\frac{\pi}{2}(\mathbf{P}_x - \mathbf{m}_x[i])}{\mathbf{d}_x[i]}$ 
7:    $y_{\text{norm}} \leftarrow \frac{\frac{\pi}{2}(\mathbf{P}_y - \mathbf{m}_y[i])}{\mathbf{d}_y[i]}$ 
8:    $h_x \leftarrow \cos(x_{\text{norm}}) \cdot e^{\frac{|x_{\text{norm}}|}{10}}$ 
9:    $h_y \leftarrow \cos(y_{\text{norm}}) \cdot e^{\frac{|y_{\text{norm}}|}{10}}$ 
10:   $h \leftarrow \mathbf{d}_z[i] \cdot h_x \cdot h_y$ 
11:  Apply boundary conditions:
12:  if  $x_{\text{norm}} > \frac{\pi}{2}$  or  $x_{\text{norm}} < -\frac{\pi}{2}$  or  $y_{\text{norm}} > \frac{\pi}{2}$  or  $y_{\text{norm}} < -\frac{\pi}{2}$  then
13:     $h \leftarrow 0$ 
14:  end if
15:   $H \leftarrow H + h$ 
16: end for
17:  $\mathbf{P} \leftarrow \text{concatenate}([\mathbf{P}, H])$ 

```

2.2 Simple Structures

2.2.1 Boxes

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

Algorithm 2 Generate Box Meshes on Sea Floor

Require: *across_track_length*, *along_track_length*, *box_coordinates*, *box_reflectivity*

- 1: **Initialize** min/max width, length, height, meshdensity, reflectivity, and number of boxes
- 2: Generate random center points for boxes:
- 3: $midxypoints \leftarrow \text{rand}([3, num_boxes])$
- 4: $midxypoints[0] \leftarrow midxypoints[0] \times across_track_length$
- 5: $midxypoints[1] \leftarrow midxypoints[1] \times along_track_length$
- 6: $midxypoints[2] \leftarrow 0$
- 7: Assign random dimensions to each box:
- 8: $boxwidths \leftarrow \text{rand}(num_boxes) \times (max_width - min_width) + min_width$
- 9: $boxlengths \leftarrow \text{rand}(num_boxes) \times (max_length - min_length) + min_length$
- 10: $boxheights \leftarrow \text{rand}(num_boxes) \times (max_height - min_height) + min_height$
- 11: **for** $i = 1$ to num_boxes **do**
- 12: Generate mesh points along each axis:
- 13: $xpoints \leftarrow \text{linspace}(-boxwidths[i]/2, boxwidths[i]/2, boxwidths[i] \times meshdensity)$
- 14: $ypoints \leftarrow \text{linspace}(-boxlengths[i]/2, boxlengths[i]/2, boxlengths[i] \times meshdensity)$
- 15: $zpoints \leftarrow \text{linspace}(0, boxheights[i], boxheights[i] \times meshdensity)$
- 16: Generate 3D mesh grid:
- 17: $X, Y, Z \leftarrow \text{meshgrid}(xpoints, ypoints, zpoints)$
- 18: Reshape X, Y, Z into 1D tensors
- 19: Compute final coordinates:
- 20: $boxcoordinates \leftarrow \text{cat}(X, Y, Z)$
- 21: $boxcoordinates[0] \leftarrow boxcoordinates[0] + midxypoints[0][i]$
- 22: $boxcoordinates[1] \leftarrow boxcoordinates[1] + midxypoints[1][i]$
- 23: $boxcoordinates[2] \leftarrow boxcoordinates[2] + midxypoints[2][i]$
- 24: Generate reflectivity values:
- 25: $boxreflectivity \leftarrow meshreflectivity + \text{rand}(1, \text{size}(boxcoordinates)) - 0.5$
- 26: Append data to final tensors:
- 27: $box_coordinates \leftarrow \text{cat}(box_coordinates, boxcoordinates, 1)$
- 28: $box_reflectivity \leftarrow \text{cat}(box_reflectivity, boxreflectivity, 1)$
- 29: **end for**

2.2.2 Sphere

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

Algorithm 3 Sphere Creation

num_hills \leftarrow Number of Hills

Chapter 3

Hardware Setup

Overview

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

3.1 Transmitter

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

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

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

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

The full-script for the setup of the transmitter is given in section ?? 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, ϕ 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, θ and the range, r .

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

3.3 Marine Vessel

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

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

Chapter 4

Signal Simulation

Overview

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

4.1 Transmitted Signal

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

4.2 Signal Simulation

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

Algorithm 4 Signal Simulation

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

4.3 Ray Tracing

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

4.3.1 Pairwise Dot-Product

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

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

4.3.2 Range Histogram Method

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

Algorithm 5 Range Histogram Method

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

Chapter 5

Imaging

Overview

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

5.1 Decimation

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

5.1.1 Basebanding

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

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

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

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

Algorithm 6 Basebanding

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

5.1.2 Lowpass filtering

1. Now that we have the signal in the baseband, we lowpass filter the signal based on the bandwidth of the signal. Since we're perfectly centering the signal using f_c , we can have the cutoff-frequency of the lowpass filter to be just above half the bandwidth of the transmitted signal. Note that the signals should not be brought down back into the real-domain using `abs()` or `real()` functions since the negative frequencies are no longer symmetrical.
2. After low-pass filtering, we have a band-restricted signal that contains all of the data in the baseband. This allows for decimation, which is what we'll do in the next step.

5.1.3 Decimation

1. Now that we have the bandlimited signal, what we shall do is decimation. Decimation essentially involves just taking every n -th sample where n in this case is the decimation factor.
2. The resulting signal contains the same information as that of the real-sampled signal but with much less number of samples.

5.2 Match-Filtering

1. To understand why match-filtering is going on, it is important to understand pulse compression.
2. In "probing" systems, which are basically systems where we send out some signal, listen to the reflection and infer quantitative and qualitative aspects of the environment, the best signal is the impulse signal (see Dirac Delta). However, this signal is not practical to use. Primarily due to the very simple fact that this particular signal has a flat and infinite bandwidth. However, this signal is the idea.
3. So instead, we're left with using signals that have a finite length, $T_{\text{Transmitted Signal}}$. However, the issue with that is that a scatter of infinitesimal dimension produce a response that has a length of $T_{\text{Transmitted Signal}}$. Thus, it is important to ensure that the response of each object, scatter or what not has comparable dimensions. This is where pulse compression comes in. Using this technique, we transform the received signal to produce a signal that is as close as possible to the signal we'd receive if we were to send out a direct delta pulse.
4. Thus, this process involves something of a detection. The closest method is something of a correlation filter where we run a copy of the transmitted signal through the received recording and take inner-products at each time step (known as the cor-

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

5. We want to produce a filter, which when convolved with the received signal produces a spike. Since we're trying to produce something similar to the response of an ideal transmission system, we want the output to be that of an ideal spike, which is the delta function. So we're essentially trying to find a filter, which when multiplied with the transmitted signal, produces the diract delta.
6. The answer can be found by analyzing the frequency domain. The frequency domain basis representation of the delta-function is a flat magnitude and linear phase. Thus, this means that the filter that we use on the transmitted signal must produce a flat magnitude and linear phase. The transmitted signal that we're working with, being an LFM, means that the magnitude is already flat. The phase, however, is quadratic. So we need the matched filter to have a flat magnitude and a quadratic phase that cancels away that of the transmitted signal's quadratic component. All this leads to the best candidate: the complex conjugate of the transmitted signal. However, since we're now working with the quadrature demodulated signal, the matched filter is the complex conjugate of the quadrature demodulated transmitted signal.
7. So once the filter is made, convolving that with the received signal produces a number of spikes in the processed signal. Note that due to working in the digital domain and some other factors, the spikes will not be perfect. Thus it is not safe to take the `abs()` or `real()` just yet. We'll do that after beamforming.
8. But so far, this marks the first step of the perception pipeline.

Algorithm 7 Match-Filtering

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

Beamforming

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

Algorithm 8 Beamforming

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

Chapter 6

Control Pipeline

Overview

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

DQN

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

Algorithm 9 DQN

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

Artificial Acoustic Imaging

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

based imaging algorithms. The function for this has been added and is available in section ?? 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 \leftarrow Coordinates of points in the point-cloud.

auvCoordinates \leftarrow Coordinates of AUV/ULA.

Chapter 7

Results

Chapter 8

Software

Overview

-

8.1 Class Definitions

8.1.1 Class: Scatter

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

```
1 // // header-files
2 // #include <iostream>
3 // #include <ostream>
4 // #include <torch/torch.h>
5
6 // #pragma once
7
8 // // hash defines
9 // #ifndef PRINTSPACE
10 // #define PRINTSPACE    std::cout<<"\n\n\n\n\n\n\n\n"<<std::endl;
11 // #endif
12 // #ifndef PRINTSMALLLINE
13 // #define PRINTSMALLLINE std::cout<<"-----"<<std::endl;
14 // #endif
15 // #ifndef PRINTLINE
16 // #define PRINTLINE    std::cout<<"===== "<<std::endl;
17 // #endif
18 // #ifndef DEVICE
19 //     #define DEVICE    torch::kMPS
20 //     // #define DEVICE    torch::kCPU
21 // #endif
22
23
24 // #define PI    3.14159265
25
26
27 // // function to print tensor size
28 // void print_tensor_size(const torch::Tensor& inputTensor) {
29 //     // Printing size
30 //     std::cout << "[";
31 //     for (const auto& size : inputTensor.sizes()) {
32 //         std::cout << size << ", ";
33 //     }
34 //     std::cout << "\b]" <<std::endl;
```

```

35 // }
36
37 // // Scatterer Class = Scatterer Class
38 // // Scatterer Class = Scatterer Class
39 // // Scatterer Class = Scatterer Class
40 // // Scatterer Class = Scatterer Class
41 // // Scatterer Class = Scatterer Class
42 // class ScattererClass{
43 // public:
44
45 //     // public variables
46 //     torch::Tensor coordinates; // tensor holding coordinates [3, x]
47 //     torch::Tensor reflectivity; // tensor holding reflectivity [1, x]
48
49 //     // constructor = constructor
50 //     ScattererClass(torch::Tensor arg_coordinates = torch::zeros({3,1}),
51 //                   torch::Tensor arg_reflectivity = torch::zeros({3,1})):
52 //         coordinates(arg_coordinates),
53 //         reflectivity(arg_reflectivity) {}
54
55 //     // overloading output
56 //     friend std::ostream& operator<<(std::ostream& os, ScattererClass& scatterer){
57
58 //         // printing coordinate shape
59 //         os<<"\t> scatterer.coordinates.shape = ";
60 //         print_tensor_size(scatterer.coordinates);
61
62 //         // printing reflectivity shape
63 //         os<<"\t> scatterer.reflectivity.shape = ";
64 //         print_tensor_size(scatterer.reflectivity);
65
66 //         // returning os
67 //         return os;
68 //     }
69
70 //     // copy constructor from a pointer
71 //     ScattererClass(ScattererClass* scatterers){
72
73 //         // copying the values
74 //         this->coordinates = scatterers->coordinates;
75 //         this->reflectivity = scatterers->reflectivity;
76 //     }
77
78 // };
79
80 template <typename T>
81 class ScattererClass
82 {
83 public:
84     // members
85     std::vector<std::vector<T>> coordinates;
86     std::vector<T> reflectivity;
87
88     // Constructor
89     ScattererClass() {}
90
91     // Constructor
92     ScattererClass(std::vector<std::vector<T>> coordinates_arg,
93                   std::vector<T> reflectivity_arg):
94         coordinates(coordinates_arg),
95         reflectivity(reflectivity_arg) {}
96
97     // Save to CSV
98     void savetocsv(){
99         fWriteMatrix(this->coordinates, "../csv-files/coordinates.csv");
100         fWriteVector(this->reflectivity, "../csv-files/reflectivity.csv");
101     }
102
103     // // overloading output
104     // friend std::ostream& operator<<(std::ostream& os, ScattererClass& scatterer){
105
106     //     // printing coordinate shape
107     //     os << format("\t> scatterer.coordinates.shape = [{}, {}]\n", scatterer.coordinates.size(),
108     //                 scatterer.coordinates[0].size());

```



```
109 // // printing reflectivity shape
110 // os << format("\t> scatterer.reflectivity.shape = [{}, {}]",
111 //             1, scatterer.reflectivity.size()) ;
112
113 // // returning os
114 // return os;
115 // }
116
117 // // copy constructor from a pointer
118 // ScattererClass(ScattererClass* scatterers){
119
120 // // copying the values
121 // this->coordinates = scatterers->coordinates;
122 // this->reflectivity = scatterers->reflectivity;
123 // }
124
125 };
```

```

69 // int azimuthQuantDensity; // quantization of angles along the azimuth
70 // int elevationQuantDensity; // quantization of angles along the elevation
71 // float rangeQuantSize; // range-cell size when shadowing
72 // float azimuthShadowThreshold; // azimuth thresholding
73 // float elevationShadowThreshold; // elevation thresholding
74
75 // // shadowing related
76 // // torch::Tensor checkbox; // box indicating whether a scatter for a range-angle pair has been
found
77 // // torch::Tensor finalScatterBox; // a 3D tensor where the third dimension represnets the vector length
78 // // torch::Tensor finalReflectivityBox; // to store the reflectivity
79
80
81
82 // // Constructor
83 // TransmitterClass(torch::Tensor location = torch::zeros({3,1}),
84 // torch::Tensor Signal = torch::zeros({10,1}),
85 // float azimuthal_angle = 0,
86 // float elevation_angle = -30,
87 // float azimuthal_beamwidth = 30,
88 // float elevation_beamwidth = 30):
89 // location(location),
90 // Signal(Signal),
91 // azimuthal_angle(azimuthal_angle),
92 // elevation_angle(elevation_angle),
93 // azimuthal_beamwidth(azimuthal_beamwidth),
94 // elevation_beamwidth(elevation_beamwidth) {}
95
96 // // overloading output
97 // friend std::ostream& operator<<(std::ostream& os, TransmitterClass& transmitter){
98 // os<<"\t> azimuth : "<<transmitter.azimuthal_angle <<std::endl;
99 // os<<"\t> elevation : "<<transmitter.elevation_angle <<std::endl;
100 // os<<"\t> azimuthal beamwidth: "<<transmitter.azimuthal_beamwidth<<std::endl;
101 // os<<"\t> elevation beamwidth: "<<transmitter.elevation_beamwidth<<std::endl;
102 // PRINTSMALLLINE
103 // return os;
104 // }
105
106 // // overloading copyign operator
107 // TransmitterClass& operator=(const TransmitterClass& other){
108
109 // // checking self-assignment
110 // if(this==&other){
111 // return *this;
112 // }
113
114 // // allocating memory
115 // this->location = other.location;
116 // this->Signal = other.Signal;
117 // this->azimuthal_angle = other.azimuthal_angle;
118 // this->elevation_angle = other.elevation_angle;
119 // this->azimuthal_beamwidth = other.azimuthal_beamwidth;
120 // this->elevation_beamwidth = other.elevation_beamwidth;
121 // this->range = other.range;
122
123 // // transmitted signal attributes
124 // this->f_low = other.f_low;
125 // this->f_high = other.f_high;
126 // this->fc = other.fc;
127 // this->bandwidth = other.bandwidth;
128
129 // // shadowing properties
130 // this->azimuthQuantDensity = other.azimuthQuantDensity;
131 // this->elevationQuantDensity = other.elevationQuantDensity;
132 // this->rangeQuantSize = other.rangeQuantSize;
133 // this->azimuthShadowThreshold = other.azimuthShadowThreshold;
134 // this->elevationShadowThreshold = other.elevationShadowThreshold;
135
136 // // this->checkbox = other.checkbox;
137 // // this->finalScatterBox = other.finalScatterBox;
138 // // this->finalReflectivityBox = other.finalReflectivityBox;
139
140 // // returning
141 // return *this;
142

```

```

143 // };
144
145 // /*=====
146 // Aim: Update pointing angle
147 // -----
148 // Note:
149 // > This function updates pointing angle based on AUV's pointing angle
150 // > for now, we're assuming no roll;
151 // -----*/
152 // void updatePointingAngle(torch::Tensor AUV_pointing_vector){
153
154 //     // calculate yaw and pitch
155 //     if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 140 \n";
156 //     torch::Tensor AUV_pointing_vector_spherical = fCart2Sph(AUV_pointing_vector);
157 //     torch::Tensor yaw = AUV_pointing_vector_spherical[0];
158 //     torch::Tensor pitch = AUV_pointing_vector_spherical[1];
159 //     if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 144 \n";
160
161 //     // std::cout<<"\t TransmitterClass: AUV_pointing_vector = "<<torch::transpose(AUV_pointing_vector,
162 //     // std::cout<<"\t TransmitterClass: AUV_pointing_vector_spherical =
163 //     // "<<torch::transpose(AUV_pointing_vector_spherical, 0, 1)<<std::endl;
164
165 //     // calculating azimuth and elevation of transmitter object
166 //     torch::Tensor absolute_azimuth_of_transmitter = yaw +
167 //     torch::tensor({this->azimuthal_angle}).to(DATATYPE).to(DEVICE);
168 //     torch::Tensor absolute_elevation_of_transmitter = pitch +
169 //     torch::tensor({this->elevation_angle}).to(DATATYPE).to(DEVICE);
170 //     if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 149 \n";
171
172 //     // std::cout<<"\t TransmitterClass: this->azimuthal_angle = "<<this->azimuthal_angle<<std::endl;
173 //     // std::cout<<"\t TransmitterClass: this->elevation_angle = "<<this->elevation_angle<<std::endl;
174 //     // std::cout<<"\t TransmitterClass: absolute_azimuth_of_transmitter =
175 //     // "<<absolute_azimuth_of_transmitter<<std::endl;
176 //     // std::cout<<"\t TransmitterClass: absolute_elevation_of_transmitter =
177 //     // "<<absolute_elevation_of_transmitter<<std::endl;
178
179 //     // converting back to Cartesian
180 //     torch::Tensor pointing_direction_spherical = torch::zeros({3,1}).to(DATATYPE).to(DEVICE);
181 //     pointing_direction_spherical[0] = absolute_azimuth_of_transmitter;
182 //     pointing_direction_spherical[1] = absolute_elevation_of_transmitter;
183 //     pointing_direction_spherical[2] = torch::tensor({1}).to(DATATYPE).to(DEVICE);
184 //     if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 60 \n";
185
186 //     this->pointing_direction = fSph2Cart(pointing_direction_spherical);
187 //     if(DEBUGMODE_TRANSMITTER) std::cout<<"\t TransmitterClass: page 169 \n";
188
189 // }
190
191 // /*=====
192 // Aim: Subsetting Scatterers inside the cone
193 // -----
194 // steps:
195 // 1. Find azimuth and range of all points.
196 // 2. Find azimuth and range of current pointing vector.
197 // 3. Subtract azimuth and range of points from that of azimuth and range of current pointing vector
198 // 4. Use tilted ellipse equation to find points in the ellipse
199 // -----*/
200 // void subsetScatterers(ScattererClass* scatterers,
201 // float tilt_angle){
202
203 //     // translationally change origin
204 //     scatterers->coordinates = \
205 //     scatterers->coordinates - this->location;
206
207 //     /*
208 //     Note: I think something we can do is see if we can subset the matrices by checking coordinate
209 //     values right away. If one of the coordinate values is x (relative coordinates), we know for sure that
210 //     the distance is greater than x, for sure. So, maybe that's something that we can work with
211 //     */
212
213 //     // Finding spherical coordinates of scatterers and pointing direction
214 //     torch::Tensor scatterers_spherical = fCart2Sph(scatterers->coordinates);

```

```

210 //      torch::Tensor pointing_direction_spherical = fCart2Sph(this->pointing_direction);
211
212
213 //      // Calculating relative azimuths and radians
214 //      torch::Tensor relative_spherical = \
215 //          scatterers_spherical - pointing_direction_spherical;
216
217
218 //      // clearing some stuff up
219 //      scatterers_spherical.reset();
220 //      pointing_direction_spherical.reset();
221
222
223 //      // tensor corresponding to switch.
224 //      torch::Tensor tilt_angle_Tensor = \
225 //          torch::tensor({tilt_angle}).to(DATATYPE).to(DEVICE);
226
227 //      // calculating length of axes
228 //      torch::Tensor axis_a = \
229 //          torch::tensor({
230 //              this->azimuthal_beamwidth / 2
231 //          }).to(DATATYPE).to(DEVICE);
232 //      torch::Tensor axis_b = \
233 //          torch::tensor({
234 //              this->elevation_beamwidth / 2
235 //          }).to(DATATYPE).to(DEVICE);
236
237 //      // part of calculating the tilted ellipse
238 //      torch::Tensor xcosa = relative_spherical[0] * torch::cos(tilt_angle_Tensor * PI/180);
239 //      torch::Tensor ysina = relative_spherical[1] * torch::sin(tilt_angle_Tensor * PI/180);
240 //      torch::Tensor xsina = relative_spherical[0] * torch::sin(tilt_angle_Tensor * PI/180);
241 //      torch::Tensor ycosa = relative_spherical[1] * torch::cos(tilt_angle_Tensor * PI/180);
242 //      relative_spherical.reset();
243
244
245 //      // finding points inside the tilted ellipse
246 //      torch::Tensor scatter_boolean = \
247 //          torch::div(torch::square(xcosa + ysina), torch::square(axis_a)) + \
248 //          torch::div(torch::square(xsina - ycosa), torch::square(axis_b)) <= 1;
249
250
251 //      // clearing
252 //      xcosa.reset(); ysina.reset(); xsina.reset(); ycosa.reset();
253
254
255 //      // subsetting points within the elliptical beam
256 //      auto mask = (scatter_boolean == 1); // creating a mask
257 //      scatterers->coordinates = scatterers->coordinates.index({torch::indexing::Slice(), mask});
258 //      scatterers->reflectivity = scatterers->reflectivity.index({torch::indexing::Slice(), mask});
259
260
261 //      // this is where histogram shadowing comes in (later)
262 //      if (ENABLE_RAYTRACING) {
263 //          rangeHistogramShadowing(scatterers);
264 //      }
265
266 //      // translating back to the points
267 //      scatterers->coordinates = scatterers->coordinates + this->location;
268
269 //  }
270
271 //  /*=====
272 //  Aim: Shadowing method (range-histogram shadowing)
273 //  .....
274 //  Note:
275 //      > cut down the number of threads into range-cells
276 //      > for each range cell, calculate histogram
277 //      >
278 //      std::cout<<"\t TransmitterClass: "
279 //      -----*/
280 //  void rangeHistogramShadowing(ScattererClass* scatterers){
281
282 //      // converting points to spherical coordinates
283 //      torch::Tensor spherical_coordinates = fCart2Sph(scatterers->coordinates); std::cout<<"\t\t
TransmitterClass: line 252 "<<std::endl;

```

```

284
285 //      // finding maximum range
286 //      torch::Tensor maxdistanceofpoints = torch::max(spherical_coordinates[2]); std::cout<<"\t\t
TransmitterClass: line 256 "<<std::endl;

287
288 //      // calculating number of range-cells (verified)
289 //      int numrangecells = std::ceil(maxdistanceofpoints.item<int>()/this->rangeQuantSize);
290
291 //      // finding range-cell boundaries (verified)
292 //      torch::Tensor rangeBoundaries = \
293 //          torch::linspace(this->rangeQuantSize, \
294 //              numrangecells * this->rangeQuantSize, \
295 //              numrangecells); std::cout<<"\t\t TransmitterClass: line 263 "<<std::endl;
296
297 //      // creating the checkbox (verified)
298 //      int numazimuthcells = std::ceil(this->azimuthal_beamwidth * this->azimuthQuantDensity);
299 //      int numelevationcells = std::ceil(this->elevation_beamwidth * this->elevationQuantDensity);
std::cout<<"\t\t TransmitterClass: line 267 "<<std::endl;

300
301 //      // finding the deltas
302 //      float delta_azimuth = this->azimuthal_beamwidth / numazimuthcells;
303 //      float delta_elevation = this->elevation_beamwidth / numelevationcells; std::cout<<"\t\t
TransmitterClass: line 271"<<std::endl;

304
305 //      // creating the centers (verified)
306 //      torch::Tensor azimuth_centers = torch::linspace(delta_azimuth/2, \
307 //          numazimuthcells * delta_azimuth - delta_azimuth/2, \
308 //          numazimuthcells);
309 //      torch::Tensor elevation_centers = torch::linspace(delta_elevation/2, \
310 //          numelevationcells * delta_elevation -
delta_elevation/2, \
311 //          numelevationcells); std::cout<<"\t\t TransmitterClass:
line 279"<<std::endl;

312
313 //      // centering (verified)
314 //      azimuth_centers = azimuth_centers + torch::tensor({this->azimuthal_angle - \
315 //          (this->azimuthal_beamwidth/2)}).to(DATATYPE);
316 //      elevation_centers = elevation_centers + torch::tensor({this->elevation_angle - \
317 //          (this->elevation_beamwidth/2)}).to(DATATYPE);
std::cout<<"\t\t TransmitterClass: line 285"<<std::endl;

318
319 //      // building checkboxes
320 //      torch::Tensor checkbox = torch::zeros({numelevationcells, numazimuthcells}, torch::kBool);
321 //      torch::Tensor finalScatterBox = torch::zeros({numelevationcells, numazimuthcells,
322 //          3}).to(DATATYPE);
323 //      torch::Tensor finalReflectivityBox = torch::zeros({numelevationcells,
numazimuthcells}).to(DATATYPE); std::cout<<"\t\t TransmitterClass: line 290"<<std::endl;

324
325 //      // going through each-range-cell
326 //      for(int i = 0; i<(int)rangeBoundaries.numel(); ++i){
327 //          this->internal_subsetCurrentRangeCell(rangeBoundaries[i], \
328 //              scatterers, \
329 //              checkbox, \
330 //              finalScatterBox, \
331 //              finalReflectivityBox, \
332 //              azimuth_centers, \
333 //              elevation_centers, \
334 //              spherical_coordinates); std::cout<<"\t\t TransmitterClass:
line 301"<<std::endl;

335
336 //      // after each-range-cell
337 //      torch::Tensor checkboxfilled = torch::sum(checkbox);
338 //      std::cout<<"\t\t\t\t checkbox-filled = "<<checkboxfilled.item<int>()<<"/"<<checkbox.numel()<<"
| percent = "<<100 * checkboxfilled.item<float>()/(float)checkbox.numel()<<std::endl;

339 //      }

340
341 //      // converting from box structure to [3, num-points] structure
342 //      torch::Tensor final_coords_spherical = \
343 //          torch::permute(finalScatterBox, {2, 0, 1}).reshape({3, (int)(finalScatterBox.numel()/3)});
344 //      torch::Tensor final_coords_cart = fSph2Cart(final_coords_spherical); std::cout<<"\t\t
TransmitterClass: line 308"<<std::endl;

345 //      std::cout<<"\t\t finalReflectivityBox.shape = "; fPrintTensorSize(finalReflectivityBox);
346 //      torch::Tensor final_reflectivity = finalReflectivityBox.reshape({finalReflectivityBox.numel()});
std::cout<<"\t\t TransmitterClass: line 310"<<std::endl;

```

```

347 //      torch::Tensor test_checkbox = checkbox.reshape({checkbox.numel()}); std::cout<<"\t\t
TransmitterClass: line 311"<<std::endl;
348
349 //      // just taking the points corresponding to the filled. Else, there's gonna be a lot of zero zero
zero tensors
350 //      auto mask = (test_checkbox == 1); std::cout<<"\t\t TransmitterClass: line 319"<<std::endl;
351 //      final_coords_cart = final_coords_cart.index({torch::indexing::Slice(), mask}); std::cout<<"\t\t
TransmitterClass: line 320"<<std::endl;
352 //      final_reflectivity = final_reflectivity.index({mask}); std::cout<<"\t\t TransmitterClass: line
321"<<std::endl;
353
354 //      // overwriting the scatterers
355 //      scatterers->coordinates = final_coords_cart;
356 //      scatterers->reflectivity = final_reflectivity; std::cout<<"\t\t TransmitterClass: line
324"<<std::endl;
357
358 //  }
359
360
361 //  void internal_subsetCurrentRangeCell(torch::Tensor rangeupperlimit, \
362 //      ScattererClass* scatterers, \
363 //      torch::Tensor& checkbox, \
364 //      torch::Tensor& finalScatterBox, \
365 //      torch::Tensor& finalReflectivityBox, \
366 //      torch::Tensor& azimuth_centers, \
367 //      torch::Tensor& elevation_centers, \
368 //      torch::Tensor& spherical_coordinates){
369
370 //      // finding indices for points in the current range-cell
371 //      torch::Tensor pointsincurrentrangeCell = \
372 //          torch::mul((spherical_coordinates[2] <= rangeupperlimit) , \
373 //              (spherical_coordinates[2] > rangeupperlimit - this->rangeQuantSize));
374
375 //      // checking out if there are no points in this range-cell
376 //      int num311 = torch::sum(pointsincurrentrangeCell).item<int>();
377 //      if(num311 == 0) return;
378
379 //      // calculating delta values
380 //      float delta_azimuth = azimuth_centers[1].item<float>() - azimuth_centers[0].item<float>();
381 //      float delta_elevation = elevation_centers[1].item<float>() - elevation_centers[0].item<float>();
382
383 //      // subsetting points in the current range-cell
384 //      auto mask = (pointsincurrentrangeCell == 1); // creating a mask
385 //      torch::Tensor reflectivityincurrentrangeCell =
scatterers->reflectivity.index({torch::indexing::Slice(), mask});
386 //      pointsincurrentrangeCell = spherical_coordinates.index({torch::indexing::Slice(),
mask});
387
388 //      // finding number of azimuth sizes and what not
389 //      int numazimuthcells = azimuth_centers.numel();
390 //      int numelevationcells = elevation_centers.numel();
391
392 //      // go through all the combinations
393 //      for(int azi_index = 0; azi_index < numazimuthcells; ++azi_index){
394 //          for(int ele_index = 0; ele_index < numelevationcells; ++ele_index){
395
396 //              // check if this particular azimuth-elevation direction has been taken-care of.
397 //              if (checkbox[ele_index][azi_index].item<bool>()) break;
398
399 //              // init (verified)
400 //              torch::Tensor current_azimuth = azimuth_centers.index({azi_index});
401 //              torch::Tensor current_elevation = elevation_centers.index({ele_index});
402
403 //              // // finding azimuth boolean
404 //              // torch::Tensor azi_neighbours = torch::abs(pointsincurrentrangeCell[0] - current_azimuth);
405 //              // azi_neighbours = azi_neighbours <= delta_azimuth; // tinker with this.
406
407 //              // // finding elevation boolean
408 //              // torch::Tensor ele_neighbours = torch::abs(pointsincurrentrangeCell[1] -
current_elevation);
409 //              // ele_neighbours = ele_neighbours <= delta_elevation;
410
411 //              // finding azimuth boolean
412 //              torch::Tensor azi_neighbours = torch::abs(pointsincurrentrangeCell[0] - current_azimuth);
413 //              azi_neighbours = azi_neighbours <= this->azimuthShadowThreshold; // tinker

```

```

with this.
414
415 //          // finding elevation boolean
416 //          torch::Tensor ele_neighbours = torch::abs(pointsincurrentrange[1] - current_elevation);
417 //          ele_neighbours          = ele_neighbours <= this->elevationShadowThreshold;
418
419
420 //          // combining booleans: means find all points that are within the limits of both the azimuth
and boolean.
421 //          torch::Tensor neighbours_boolean = torch::mul(azi_neighbours, ele_neighbours);
422
423 //          // checking if there are any points along this direction
424 //          int num347 = torch::sum(neighbours_boolean).item<int>();
425 //          if (num347 == 0) continue;
426
427 //          // findings point along this direction
428 //          mask          = (neighbours_boolean == 1);
429 //          torch::Tensor coords_along_aziele_spherical =
pointsincurrentrange.index({torch::indexing::Slice(), mask});
430 //          torch::Tensor reflectivity_along_aziele =
reflectivityincurrentrange.index({torch::indexing::Slice(), mask});
431
432 //          // finding the index where the points are at the maximum distance
433 //          int index_where_min_range_is = torch::argmin(coords_along_aziele_spherical[2]).item<int>();
434 //          torch::Tensor closest_coord = coords_along_aziele_spherical.index({torch::indexing::Slice(),
\
435 //          index_where_min_range_is});
436 //          torch::Tensor closest_reflectivity =
reflectivity_along_aziele.index({torch::indexing::Slice(), \
437 //          index_where_min_range_is});
438
439 //          // filling the matrices up
440 //          finalScatterBox.index_put_({ele_index, azi_index, torch::indexing::Slice()}, \
441 //          closest_coord.reshape({1,1,3}));
442 //          finalReflectivityBox.index_put_({ele_index, azi_index}, \
443 //          closest_reflectivity);
444 //          checkbox.index_put_({ele_index, azi_index}, \
445 //          true);
446
447 //      }
448 //  }
449 // }
450
451
452
453
454 // };
455
456
457
458 template <typename T>
459 class TransmitterClass{
460 public:
461
462     // physical/intrinsic properties
463     std::vector<T>    location;          // location tensor
464     std::vector<T>    pointing_direction; // pointing direction
465
466     // basic parameters
467     std::vector<T>    Signal;          // transmitted signal (LFM)
468     T                  azimuthal_angle; // transmitter's azimuthal pointing direction
469     T                  elevation_angle;  // transmitter's elevation pointing direction
470     T                  azimuthal_beamwidth; // azimuthal beamwidth of transmitter
471     T                  elevation_beamwidth; // elevation beamwidth of transmitter
472     T                  range;           // a parameter used for spotlight mode.
473
474     // transmitted signal attributes
475     T                  f_low;           // lowest frequency of LFM
476     T                  f_high;          // highest frequency of LFM
477     T                  fc;              // center frequency of LFM
478     T                  bandwidth;       // bandwidth of LFM
479
480     // shadowing properties
481     int                azimuthQuantDensity; // quantization of angles along the azimuth
482     int                elevationQuantDensity; // quantization of angles along the elevation

```



```

483     T                rangeQuantSize;           // range-cell size when shadowing
484     T                azimuthShadowThreshold;    // azimuth thresholding
485     T                elevationShadowThreshold;  // elevation thresholding
486
487     // shadowing related
488     std::vector<T>    checkbox;                  // box indicating whether a scatter for a range-angle pair has been
489                                           found
490     std::vector<std::vector<std::vector<T>>> finalScatterBox; // a 3D tensor where the third dimension
491                                           represnets the vector length
492     std::vector<T>    finalReflectivityBox; // to store the reflectivity
493
494     // constructor
495     TransmitterClass() = default;
496
497     // Deleting copy constructors/assignment
498     TransmitterClass(const TransmitterClass& other) = delete;
499     TransmitterClass& operator=(TransmitterClass& other) = delete;
500
501     // Creating move-constructor and move-assignment
502     TransmitterClass(TransmitterClass&& other) = default;
503     TransmitterClass& operator=(TransmitterClass&& other) = default;
504
505     /*=====
506     Aim: Update pointing angle
507     -----
508     Note:
509     > This function updates pointing angle based on AUV's pointing angle
510     > for now, we're assuming no roll;
511     -----*/
512     auto updatePointingAngle(std::vector<T> AUV_pointing_vector);
513 };
514
515 /*=====
516 Aim: Update pointing angle
517 -----
518 Note:
519 > This function updates pointing angle based on AUV's pointing angle
520 > for now, we're assuming no roll;
521 -----*/
522
523 template <typename T>
524 auto TransmitterClass<T>::updatePointingAngle(std::vector<T> AUV_pointing_vector)
525 {
526     // calculate yaw and pitch
527     auto AUV_pointing_vector_spherical {svr::cart2sph(AUV_pointing_vector)};
528     auto yaw {AUV_pointing_vector_spherical[0]};
529     auto pitch {AUV_pointing_vector_spherical[1]};
530
531     // calculating azimuth and elevation of transmitter object
532     auto absolute_azimuth_of_transmitter {yaw + this->azimuthal_angle};
533     auto absolute_elevation_of_transmitter {pitch + this->elevation_angle};
534
535     // converting back to Cartesian
536     auto pointing_direction_spherical {std::vector<T>(3, 0)};
537     pointing_direction_spherical[0] = absolute_azimuth_of_transmitter;
538     pointing_direction_spherical[1] = absolute_elevation_of_transmitter;
539     pointing_direction_spherical[2] = 1;
540     this->pointing_direction = svr::sph2cart(pointing_direction_spherical);
541 }

```

```

66 // int decimation_factor; // the new decimation factor
67 // float post_decimation_sampling_frequency; // the new sampling frequency
68 // torch::Tensor lowpassFilterCoefficientsForDecimation; //
69
70 // // imaging related
71 // float range_resolution; // theoretical range-resolution =  $\frac{c}{2B}$ 
72 // float azimuthal_resolution; // theoretical azimuth-resolution =
 $\frac{\lambda}{(N-1) \cdot \text{inter-element-distance}}$ 
73 // float range_cell_size; // the range-cell quanta we're choosing for efficiency trade-off
74 // float azimuth_cell_size; // the azimuth quanta we're choosing
75 // torch::Tensor mulFFTMatrix; // the matrix containing the delays for each-element as a slot
76 // torch::Tensor azimuth_centers; // tensor containing the azimuth centers
77 // torch::Tensor range_centers; // tensor containing the range-centers
78 // int frame_size; // the frame-size corresponding to a range cell in a decimated signal
matrix
79 // torch::Tensor matchFilter; // torch tensor containing the match-filter
80 // int num_buffer_zeros_per_frame; // number of zeros we're adding per frame to ensure no-rotation
81 // torch::Tensor beamformedImage; // the beamformed image
82 // torch::Tensor cartesianImage;
83
84 // // artificial acoustic-image related
85 // torch::Tensor currentArtificialAcousticImage; // the acoustic image directly produced
86
87 // // constructor
88 // ULAClass(int numsensors = 32,
89 // float inter_element_spacing = 1e-3,
90 // torch::Tensor coordinates = torch::zeros({3, 2}),
91 // float sampling_frequency = 48e3,
92 // float recording_period = 1,
93 // torch::Tensor location = torch::zeros({3,1}),
94 // torch::Tensor signalMatrix = torch::zeros({1, 32}),
95 // torch::Tensor lowpassFilterCoefficientsForDecimation = torch::zeros({1,10})):
96 // num_sensors(numsensors),
97 // inter_element_spacing(inter_element_spacing),
98 // coordinates(coordinates),
99 // sampling_frequency(sampling_frequency),
100 // recording_period(recording_period),
101 // location(location),
102 // signalMatrix(signalMatrix),
103 // lowpassFilterCoefficientsForDecimation(lowpassFilterCoefficientsForDecimation){
104 // // calculating ULA direction
105 // torch::Tensor sensorDirection = coordinates.slice(1, 0, 1) - coordinates.slice(1, 1, 2);
106
107 // // normalizing
108 // float normvalue = torch::linalg_norm(sensorDirection, 2, 0, true,
torch::kFloat).item<float>();
109
110
111 // if (normvalue != 0){
112 // sensorDirection = sensorDirection / normvalue;
113 // }
114
115 // // copying direction
116 // this->sensorDirection = sensorDirection.to(DATATYPE);
117 // }
118
119 // // overriding printing
120 // friend std::ostream& operator<<(std::ostream& os, ULAClass& ula){
121 // os<<"\t number of sensors : "<<ula.num_sensors <<std::endl;
122 // os<<"\t inter-element spacing: "<<ula.inter_element_spacing <<std::endl;
123 // os<<"\t sensor-direction " <<torch::transpose(ula.sensorDirection, 0, 1)<<std::endl;
124 // PRINTSMALLLINE
125 // return os;
126 // }
127
128 // /* =====
129 // Aim: Init
130 // ----- */
131 // void init(TransmitterClass* transmitterObj){
132
133 // // calculating range-related parameters
134 // this->range_resolution = 1500/(2 * transmitterObj->fc);
135 // this->range_cell_size = 40 * this->range_resolution;
136 // if (DEBUG_ULA) std::cout << "\t ULAClass::init: line 136" << std::endl;
137

```

```

138 //      // status printing
139 //      if (DEBUG_ULA) {
140 //          std::cout << "\t\t ULAClass::init(): this->range_resolution = " \
141 //              << this->range_resolution \
142 //              << std::endl;
143 //          std::cout << "\t\t ULAClass::init(): this->range_cell_size = " \
144 //              << this->range_cell_size \
145 //              << std::endl;
146 //      }
147 //      if (DEBUG_ULA) std::cout << "\t ULAClass::init: line 147" << std::endl;
148
149 //      // calculating azimuth-related parameters
150 //      this->azimuthal_resolution = \
151 //          (1500/transmitterObj->fc) \
152 //          /((this->num_sensors-1)*this->inter_element_spacing);
153 //      this->azimuth_cell_size = 2 * this->azimuthal_resolution;
154 //      if (DEBUG_ULA) std::cout << "\t ULAClass::init: line 154" << std::endl;
155
156 //      // creating and storing the match-filter
157 //      this->nfdc_CreateMatchFilter(transmitterObj);
158 //      if (DEBUG_ULA) std::cout << "\t ULAClass::init: line 158" << std::endl;
159 //  }
160
161 //  // Create match-filter
162 //  void nfdc_CreateMatchFilter(TransmitterClass* transmitterObj){
163
164 //      // creating matrix for basebanding the signal
165 //      torch::Tensor basebanding_vector = \
166 //          torch::linspace( \
167 //              0, \
168 //              transmitterObj->Signal.numel()-1, \
169 //              transmitterObj->Signal.numel() \
170 //          ).reshape(transmitterObj->Signal.sizes());
171 //      basebanding_vector = \
172 //          torch::exp( \
173 //              -1 * COMPLEX_1j * 2 * PI \
174 //              * (transmitterObj->fc/this->sampling_frequency) \
175 //              * basebanding_vector);
176 //      if (DEBUG_ULA) std::cout << "\t\t ULAClass::nfdc_createMatchFilter: line 176" << std::endl;
177
178 //      // multiplying the signal with the basebanding vector
179 //      torch::Tensor match_filter = \
180 //          torch::mul(transmitterObj->Signal, \
181 //              basebanding_vector);
182 //      if (DEBUG_ULA) std::cout << "\t\t ULAClass::nfdc_createMatchFilter: line 182" << std::endl;
183
184 //      // low-pass filtering to get the baseband signal
185 //      fConvolve1D(match_filter, this->lowpassFilterCoefficientsForDecimation);
186 //      if (DEBUG_ULA) std::cout << "\t\t ULAClass::nfdc_createMatchFilter: line 186" << std::endl;
187
188 //      // creating sampling-indices
189 //      int decimation_factor = \
190 //          std::floor((static_cast<float>(this->sampling_frequency)/2) \
191 //              /(static_cast<float>(transmitterObj->bandwidth)/2));
192 //      int final_num_samples = \
193 //          std::ceil(static_cast<float>(match_filter.numel())/static_cast<float>(decimation_factor));
194 //      torch::Tensor sampling_indices = \
195 //          torch::linspace(1, \
196 //              (final_num_samples-1) * decimation_factor, \
197 //              final_num_samples).to(torch::kInt) - torch::tensor({1}).to(torch::kInt);
198 //      if (DEBUG_ULA) std::cout << "ULAClass::nfdc_createMatchFilter: line 197" << std::endl;
199
200 //      // sampling the signal
201 //      match_filter = match_filter.index({sampling_indices});
202
203 //      // taking conjugate and flipping the signal
204 //      match_filter = torch::flipud( match_filter);
205 //      match_filter = torch::conj( match_filter);
206
207 //      // storing the match-filter to the class member
208 //      this->matchFilter = match_filter;
209 //  }
210
211 //  // overloading the "=" operator
212 //  ULAClass& operator=(const ULAClass& other){

```

```

213 //      // checking if copying to the same object
214 //      if(this == &other){
215 //          return *this;
216 //      }
217
218 //      // copying everything
219 //      this->num_sensors      = other.num_sensors;
220 //      this->inter_element_spacing = other.inter_element_spacing;
221 //      this->coordinates      = other.coordinates.clone();
222 //      this->sampling_frequency = other.sampling_frequency;
223 //      this->recording_period  = other.recording_period;
224 //      this->sensorDirection   = other.sensorDirection.clone();
225
226 //      // new additions
227 //      // this->location          = other.location;
228 //      this->lowpassFilterCoefficientsForDecimation = other.lowpassFilterCoefficientsForDecimation;
229 //      // this->sensorDirection   = other.sensorDirection.clone();
230 //      // this->signalMatrix      = other.signalMatrix.clone();
231
232
233 //      // returning
234 //      return *this;
235 //  }
236
237 //  // build sensor-coordinates based on location
238 //  void buildCoordinatesBasedOnLocation(){
239
240 //      // length-normalize the sensor-direction
241 //      this->sensorDirection = torch::div(this->sensorDirection, torch::linalg_norm(this->sensorDirection,
242 // \
243 //                                     2, 0, true, \
244 //                                     DATATYPE));
245
246 //      if(DEBUG_ULA) std::cout<<"\t ULAClass: line 105 \n";
247
248 //      // multiply with inter-element distance
249 //      this->sensorDirection = this->sensorDirection * this->inter_element_spacing;
250 //      this->sensorDirection = this->sensorDirection.reshape({this->sensorDirection.numel(), 1});
251 //      if(DEBUG_ULA) std::cout<<"\t ULAClass: line 110 \n";
252
253 //      // create integer-array
254 //      // torch::Tensor integer_array = torch::linspace(0, \
255 //      //                                     this->num_sensors-1, \
256 //      //                                     this->num_sensors).reshape({1,
257 //      //                                     this->num_sensors}).to(DATATYPE);
258 //      torch::Tensor integer_array = torch::linspace(0, \
259 //      //                                     this->num_sensors-1, \
260 //      //                                     this->num_sensors).reshape({1, \
261 //      //                                     this->num_sensors});
262 //      std::cout<<"integer_array = "; fPrintTensorSize(integer_array);
263 //      if(DEBUG_ULA) std::cout<<"\t ULAClass: line 116 \n";
264
265 //      //
266 //      torch::Tensor test = torch::mul(torch::tile(integer_array, {3, 1}).to(DATATYPE), \
267 //      //      torch::tile(this->sensorDirection, {1,
268 //      //      this->num_sensors}).to(DATATYPE));
269 //      this->coordinates = this->location + test;
270 //      if(DEBUG_ULA) std::cout<<"\t ULAClass: line 120 \n";
271
272 //  }
273
274 //  // signal simulation for the current sensor-array
275 //  void nfdc_simulateSignal(ScattererClass* scatterers,
276 //  //      TransmitterClass* transmitterObj){
277
278 //      // creating signal matrix
279 //      int numsamples      = std::ceil((this->sampling_frequency * this->recording_period));
280 //      this->signalMatrix = torch::zeros({numsamples, this->num_sensors}).to(DATATYPE);
281
282
283 //      // getting shape of coordinates
284 //      std::vector<int64_t> scatterers_coordinates_shape = \
285 //      scatterers->coordinates.sizes().vec();
286
287 //      // making a slot out of the coordinates
288 //      torch::Tensor slottedCoordinates = \

```

```

285 //      torch::permute(scatterers->coordinates.reshape({
286 //          scatterers_coordinates_shape[0],          \
287 //          scatterers_coordinates_shape[1],          \
288 //          1})                                       \
289 //      ), {2, 1, 0}).reshape({
290 //          1,                                         \
291 //          (int)(scatterers->coordinates.numel()/3), \
292 //          3});
293
294
295 //      // repeating along the y-direction number of sensor times.
296 //      slottedCoordinates =
297 //          torch::tile(slottedCoordinates,          \
298 //              {this->num_sensors, 1, 1});
299 //      std::vector<int64_t> slottedCoordinates_shape = \
300 //          slottedCoordinates.sizes().vec();
301
302
303 //      // finding the shape of the sensor-coordinates
304 //      std::vector<int64_t> sensor_coordinates_shape = \
305 //          this->coordinates.sizes().vec();
306
307 //      // creating a slot tensor out of the sensor-coordinates
308 //      torch::Tensor slottedSensors = \
309 //          torch::permute(this->coordinates.reshape({
310 //              sensor_coordinates_shape[0],          \
311 //              sensor_coordinates_shape[1],          \
312 //              1}), {2, 1, 0}).reshape({(int)(this->coordinates.numel()/3), \
313 //              1,                                     \
314 //              3});
315
316
317 //      // repeating slices along the x-coordinates
318 //      slottedSensors = \
319 //          torch::tile(slottedSensors,          \
320 //              {1, slottedCoordinates_shape[1], 1});
321
322 //      // slotting the coordinate of the transmitter and duplicating along dimensions [0] and [1]
323 //      torch::Tensor slotted_location = \
324 //          torch::permute(this->location.reshape({3, 1, 1}), \
325 //              {2, 1, 0}).reshape({1,1,3});
326 //      slotted_location = \
327 //          torch::tile(slotted_location, {slottedCoordinates_shape[0], \
328 //              slottedCoordinates_shape[1], \
329 //              1});
330
331
332
333 //      // subtracting to find the relative distances
334 //      torch::Tensor distBetweenScatterersAndSensors = \
335 //          torch::linalg_norm(slottedCoordinates - slottedSensors, \
336 //              2, 2, true, torch::kFloat).to(DATATYPE);
337
338 //      // subtracting distance between relative fields
339 //      torch::Tensor distBetweenScatterersAndTransmitter = \
340 //          torch::linalg_norm(slottedCoordinates - slotted_location, \
341 //              2, 2, true, torch::kFloat).to(DATATYPE);
342
343 //      // adding up the distances
344 //      torch::Tensor distOfFlight = \
345 //          distBetweenScatterersAndSensors + distBetweenScatterersAndTransmitter;
346 //      torch::Tensor timeOfFlight = distOfFlight/1500;
347 //      torch::Tensor samplesOfFlight = \
348 //          torch::floor(timeOfFlight.squeeze() \
349 //              * this->sampling_frequency);
350
351
352
353 //      // Adding pulses
354 //      #pragma omp parallel for
355 //      for(int sensor_index = 0; sensor_index < this->num_sensors; ++sensor_index){
356 //          for(int scatter_index = 0; scatter_index < samplesOfFlight[0].numel(); ++scatter_index){
357
358 //              // getting the sample where the current scatter's contribution must be placed.
359 //              int where_to_place = \

```

```

360 //          samplesOfFlight.index({sensor_index, \
361 //                                scatter_index \
362 //                                }).item<int>());
363
364 //          // checking whether that point is out of bounds
365 //          if(where_to_place >= numsamples) continue;
366
367 //          // placing a reflectivity-scaled impulse in there
368 //          this->signalMatrix.index_put_({where_to_place, sensor_index}, \
369 //          this->signalMatrix.index({where_to_place, \
370 //          sensor_index}) + \
371 //          scatterers->reflectivity.index({0, \
372 //          scatter_index})));
373 //      }
374 //  }
375
376
377
378 //  // Adding pulses
379 //  // for(int sensor_index = 0; sensor_index < this->num_sensors; ++sensor_index){
380
381 //      // indices associated with current index
382 //      torch::Tensor tensor_containing_placing_indices = \
383 //      samplesOfFlight[sensor_index].to(torch::kInt);
384
385 //      // calculating histogram
386 //      auto uniqueOutputs = at::_unique(tensor_containing_placing_indices, false, true);
387 //      torch::Tensor bruh = std::get<1>(uniqueOutputs);
388 //      torch::Tensor uniqueValues = std::get<0>(uniqueOutputs).to(torch::kInt);
389 //      torch::Tensor uniqueCounts = torch::bincount(bruh).to(torch::kInt);
390
391 //      // placing values according to histogram
392 //      this->signalMatrix.index_put_({uniqueValues.to(torch::kLong), sensor_index}, \
393 //      uniqueCounts.to(DATATYPE));
394
395 //  }
396
397 //  // Creating matrix out of transmitted signal
398 //  torch::Tensor signalTensorAsArgument = \
399 //  transmitterObj->Signal.reshape({
400 //  transmitterObj->Signal.numel(), \
401 //  1});
402 //  signalTensorAsArgument = \
403 //  torch::tile(signalTensorAsArgument, \
404 //  {1, this->signalMatrix.size(1)});
405
406
407
408 //  // convolving the pulse-matrix with the signal matrix
409 //  fConvolveColumns(this->signalMatrix, \
410 //  signalTensorAsArgument);
411
412
413 //  // trimming the convolved signal since the signal matrix length remains the same
414 //  this->signalMatrix = \
415 //  this->signalMatrix.index({
416 //  torch::indexing::Slice(0, numsamples), \
417 //  torch::indexing::Slice()});
418
419
420 //  // returning
421 //  return;
422 //  }
423
424 //  /* =====
425 //  Aim: Decimating basebanded-received signal
426 //  ===== */
427 //  void nfdc_decimateSignal(TransmitterClass* transmitterObj){
428
429 //      // creating the matrix for frequency-shifting
430 //      torch::Tensor integerArray = torch::linspace(0, \
431 //      this->signalMatrix.size(0)-1, \
432 //
433 //      this->signalMatrix.size(0)).reshape({this->signalMatrix.size(0), 1});
434 //      integerArray = torch::tile(integerArray, {1, this->num_sensors});

```

```

434 //      integerArray          = torch::exp(COMPLEX_1j * transmitterObj->fc * integerArray);
435
436 //      // storing original number of samples
437 //      int original_signalMatrix_numsamples = this->signalMatrix.size(0);
438
439 //      // producing frequency-shifting
440 //      this->signalMatrix      = torch::mul(this->signalMatrix, integerArray);
441
442 //      // low-pass filter
443 //      torch::Tensor lowpassfilter_impulseresponse = \
444 //          this->lowpassFilterCoefficientsForDecimation.reshape(\
445 //              {this->lowpassFilterCoefficientsForDecimation.numel(), \
446 //              1});
447 //      lowpassfilter_impulseresponse = \
448 //          torch::tile(lowpassfilter_impulseresponse, \
449 //              {1, this->signalMatrix.size(1)});
450
451 //      // low-pass filtering the signal
452 //      fConvolveColumns(this->signalMatrix,
453 //          lowpassfilter_impulseresponse);
454
455 //      // Cutting down the extra-samples from convolution
456 //      this->signalMatrix = \
457 //          this->signalMatrix.index({torch::indexing::Slice(0, original_signalMatrix_numsamples), \
458 //              torch::indexing::Slice()});
459
460 //      // // Cutting off samples in the front.
461 //      // int cutoffpoint = lowpassfilter_impulseresponse.size(0) - 1;
462 //      // this->signalMatrix = \
463 //      //      this->signalMatrix.index({ \
464 //      //          torch::indexing::Slice(cutoffpoint, \
465 //      //              torch::indexing::None), \
466 //      //          torch::indexing::Slice() \
467 //      //      });
468
469 //      // building parameters for downsampling
470 //      int decimation_factor      = std::floor(this->sampling_frequency/transmitterObj->bandwidth);
471 //      this->decimation_factor      = decimation_factor;
472 //      this->post_decimation_sampling_frequency = \
473 //          this->sampling_frequency / this->decimation_factor;
474 //      int numsamples_after_decimation = std::floor(this->signalMatrix.size(0)/decimation_factor);
475
476 //      // building the samples which will be subsetted
477 //      torch::Tensor samplingIndices = \
478 //          torch::linspace(0, \
479 //              numsamples_after_decimation * decimation_factor - 1, \
480 //              numsamples_after_decimation).to(torch::kInt);
481
482 //      // downsampling the low-pass filtered signal
483 //      this->signalMatrix = \
484 //          this->signalMatrix.index({samplingIndices, \
485 //              torch::indexing::Slice()});
486
487 //      // returning
488 //      return;
489 //  }
490
491 //  /* =====
492 //  Aim: Match-filtering
493 //  ----- */
494 //  void nfdc_matchFilterDecimatedSignal(){
495
496 //      // Creating a 2D matrix out of the signal
497 //      torch::Tensor matchFilter2DMatrix = \
498 //          this->matchFilter.reshape({this->matchFilter.numel(), 1});
499 //      matchFilter2DMatrix = \
500 //          torch::tile(matchFilter2DMatrix, \
501 //              {1, this->num_sensors});
502
503
504 //      // 2D convolving to produce the match-filtering
505 //      fConvolveColumns(this->signalMatrix, \
506 //          matchFilter2DMatrix);
507
508

```



```

509 //      // Trimming the signal to contain just the signals that make sense to us
510 //      int startingpoint = matchFilter2DMatrix.size(0) - 1;
511 //      this->signalMatrix = \
512 //          this->signalMatrix.index({ \
513 //              torch::indexing::Slice(startingpoint, \
514 //                                      torch::indexing::None), \
515 //              torch::indexing::Slice()});
516
517 //      // // trimming the two ends of the signal
518 //      // int startingpoint = matchFilter2DMatrix.size(0) - 1;
519 //      // int endingpoint = this->signalMatrix.size(0) \
520 //          - matchFilter2DMatrix.size(0) \
521 //          + 1;
522 //      // this->signalMatrix = \
523 //          // this->signalMatrix.index({ \
524 //              // torch::indexing::Slice(startingpoint, \
525 //                  // endingpoint), \
526 //              // torch::indexing::Slice()});
527
528
529 // }
530
531 // /* =====
532 // Aim: precompute delay-matrices
533 // ----- */
534 // void nfdc_precomputeDelayMatrices(TransmitterClass* transmitterObj){
535
536 //     // calculating range-related parameters
537 //     int number_of_range_cells = \
538 //         std::ceil(((this->recording_period * 1500)/2)/this->range_cell_size);
539 //     int number_of_azimuths_to_image = \
540 //         std::ceil(transmitterObj->azimuthal_beamwidth / this->azimuth_cell_size);
541
542 //     // creating centers of range-cell centers
543 //     torch::Tensor range_centers = \
544 //         this->range_cell_size * \
545 //         torch::arange(0, number_of_range_cells) \
546 //         + this->range_cell_size/2;
547 //     this->range_centers = range_centers;
548
549 //     // creating discretized azimuth-centers
550 //     torch::Tensor azimuth_centers = \
551 //         this->azimuth_cell_size * \
552 //         torch::arange(0, number_of_azimuths_to_image) \
553 //         + this->azimuth_cell_size/2;
554 //     this->azimuth_centers = azimuth_centers;
555 //     this->azimuth_centers = this->azimuth_centers - torch::mean(this->azimuth_centers);
556
557 //     // finding the mesh values
558 //     auto range_azimuth_meshgrid = \
559 //         torch::meshgrid({range_centers, \
560 //             azimuth_centers}, "ij");
561 //     torch::Tensor range_grid = range_azimuth_meshgrid[0]; // the columns are range_centers
562 //     torch::Tensor azimuth_grid = range_azimuth_meshgrid[1]; // the rows are azimuth-centers
563
564 //     // going from 2D to 3D
565 //     range_grid = \
566 //         torch::tile(range_grid.reshape({range_grid.size(0), \
567 //             range_grid.size(1), \
568 //             1}), {1,1,this->num_sensors});
569 //     azimuth_grid = \
570 //         torch::tile(azimuth_grid.reshape({azimuth_grid.size(0), \
571 //             azimuth_grid.size(1), \
572 //             1}), {1, 1, this->num_sensors});
573
574 //     // creating x_m tensor
575 //     torch::Tensor sensorCoordinatesSlot = \
576 //         this->inter_element_spacing * \
577 //         torch::arange(0, this->num_sensors).reshape({
578 //             1, 1, this->num_sensors
579 //         }).to(DATATYPE);
580
581 //     sensorCoordinatesSlot = \
582 //         torch::tile(sensorCoordinatesSlot, \
583 //             {range_grid.size(0),\

```

```

584 //          range_grid.size(1),
585 //          1});
586
587 //      // calculating distances
588 //      torch::Tensor distanceMatrix = \
589 //          torch::square(range_grid - sensorCoordinatesSlot) + \
590 //          torch::mul((2 * sensorCoordinatesSlot), \
591 //          torch::mul(range_grid, \
592 //          1 - torch::cos(azimuth_grid * PI/180)));
593 //      distanceMatrix = torch::sqrt(distanceMatrix);
594
595 //      // finding the time taken
596 //      torch::Tensor timeMatrix = distanceMatrix/1500;
597 //      torch::Tensor sampleMatrix = timeMatrix * this->sampling_frequency;
598
599 //      // finding the delay to be given
600 //      auto bruh390 = torch::max(sampleMatrix, 2, true);
601 //      torch::Tensor max_delay = std::get<0>(bruh390);
602 //      torch::Tensor delayMatrix = max_delay - sampleMatrix;
603
604 //      // now that we have the delay entries, we need to create the matrix that does it
605 //      int decimation_factor = \
606 //          std::floor(static_cast<float>(this->sampling_frequency)/transmitterObj->bandwidth);
607 //      this->decimation_factor = decimation_factor;
608
609
610 //      // calculating frame-size
611 //      int frame_size = \
612 //          std::ceil(static_cast<float>((2 * this->range_cell_size / 1500) * \
613 //          static_cast<float>(this->sampling_frequency)/decimation_factor));
614 //      this->frame_size = frame_size;
615
616 //      // // calculating the buffer-zeros to add
617 //      // int num_buffer_zeros_per_frame = \
618 //      //     static_cast<float>(this->num_sensors - 1) * \
619 //      //     static_cast<float>(this->inter_element_spacing) * \
620 //      //     this->sampling_frequency /1500;
621
622 //      int num_buffer_zeros_per_frame = \
623 //          std::ceil((this->num_sensors - 1) * \
624 //          this->inter_element_spacing * \
625 //          this->sampling_frequency \
626 //          / (1500 * this->decimation_factor));
627
628 //      // storing to class member
629 //      this->num_buffer_zeros_per_frame = \
630 //          num_buffer_zeros_per_frame;
631
632 //      // calculating the total frame-size
633 //      int total_frame_size = \
634 //          this->frame_size + this->num_buffer_zeros_per_frame;
635
636 //      // creating the multiplication matrix
637 //      torch::Tensor mulFFTMMatrix = \
638 //          torch::linspace(0, \
639 //          total_frame_size-1, \
640 //          total_frame_size).reshape({1, \
641 //          total_frame_size, \
642 //          1}).to(DATATYPE); // creating an array 1,...,frame_size
643 //      of shape [1,frame_size, 1];
644 //      mulFFTMMatrix = \
645 //          torch::div(mulFFTMMatrix, \
646 //          torch::tensor(total_frame_size).to(DATATYPE)); // dividing by N
647 //      mulFFTMMatrix = mulFFTMMatrix * 2 * PI * -1 * COMPLEX_1j; // creating tenosr values for -1j * 2pi *
648 //      k/N
649 //      mulFFTMMatrix = \
650 //          torch::tile(mulFFTMMatrix, \
651 //          {number_of_range_cells * number_of_azimuths_to_image, \
652 //          1, \
653 //          this->num_sensors}); // creating the larger tensor for it
654
655 //      // populating the matrix
656 //      for(int azimuth_index = 0; \
657 //          azimuth_index<number_of_azimuths_to_image; \

```

```

657 //      ++azimuth_index){
658 //      for(int range_index = 0; \
659 //      range_index < number_of_range_cells; \
660 //      ++range_index){
661 //      // finding the delays for sensors
662 //      torch::Tensor currentSensorDelays = \
663 //      delayMatrix.index({range_index, \
664 //      azimuth_index, \
665 //      torch::indexing::Slice()});
666 //      // reshaping it to the target size
667 //      currentSensorDelays = \
668 //      currentSensorDelays.reshape({1, \
669 //      1, \
670 //      this->num_sensors});
671 //      // tiling across the plane
672 //      currentSensorDelays = \
673 //      torch::tile(currentSensorDelays, \
674 //      {1, total_frame_size, 1});
675 //      // multiplying across the appropriate plane
676 //      int index_to_place_at = \
677 //      azimuth_index * number_of_range_cells + \
678 //      range_index;
679 //      mulFFTMMatrix.index_put_({index_to_place_at, \
680 //      torch::indexing::Slice(),
681 //      torch::indexing::Slice()}, \
682 //      currentSensorDelays);
683 //      }
684 //      }
685
686 //      // storing the mulFFTMMatrix
687 //      this->mulFFTMMatrix = mulFFTMMatrix;
688 //      }
689
690 //      /* =====
691 //      Aim: Beamforming the signal
692 //      ----- */
693 //      void nfdc_beamforming(TransmitterClass* transmitterObj){
694
695 //      // ensuring the signal matrix is in the shape we want
696 //      if(this->signalMatrix.size(1) != this->num_sensors)
697 //      throw std::runtime_error("The second dimension doesn't correspond to the number of sensors \n");
698
699 //      // adding the batch-dimension
700 //      this->signalMatrix = \
701 //      this->signalMatrix.reshape({
702 //      1,
703 //      this->signalMatrix.size(0),
704 //      this->signalMatrix.size(1)});
705
706
707 //      // zero-padding to ensure correctness
708 //      int ideal_length = \
709 //      std::ceil(this->range_centers.numel() * this->frame_size);
710 //      int num_zeros_to_pad_signal_along_dimension_0 = \
711 //      ideal_length - this->signalMatrix.size(1);
712
713
714 //      // printing
715 //      if (DEBUG_ULA) PRINTSMALLLINE
716 //      if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->range_centers.numel()   =
717 //      "<<this->range_centers.numel() <<std::endl;
718 //      if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->frame_size         =
719 //      "<<this->frame_size <<std::endl;
720 //      if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | ideal_length           =
721 //      "<<ideal_length <<std::endl;
722 //      if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->signalMatrix.size(1)   =
723 //      "<<this->signalMatrix.size(1) <<std::endl;
724 //      if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming |
725 //      num_zeros_to_pad_signal_along_dimension_0 = "<<num_zeros_to_pad_signal_along_dimension_0 <<std::endl;
726 //      if (DEBUG_ULA) PRINTSPACE
727
728 //      // appending or slicing based on the requirements
729 //      if (num_zeros_to_pad_signal_along_dimension_0 <= 0) {
730
731 //      // sending out a warning that slicing is going on

```

```

727 //         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;
728
729 //         // slicing the signal matrix
730 //         if (DEBUG_ULA) PRINTSPACE
731 //         if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->signalMatrix.shape (before
slicing) = "<< this->signalMatrix.sizes().vec() <<std::endl;
732 //         this->signalMatrix = \
733 //             this->signalMatrix.index({torch::indexing::Slice(), \
734 //                                     torch::indexing::Slice(0, ideal_length), \
735 //                                     torch::indexing::Slice()});
736 //         if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming | this->signalMatrix.shape (after
slicing) = "<< this->signalMatrix.sizes().vec() <<std::endl;
737 //         if (DEBUG_ULA) PRINTSPACE
738
739 //     }
740 //     else {
741 //         // creating a zero-filled tensor to append to signal matrix
742 //         torch::Tensor zero_tensor = \
743 //             torch::zeros({this->signalMatrix.size(0), \
744 //                           num_zeros_to_pad_signal_along_dimension_0, \
745 //                           this->num_sensors}).to(DATATYPE);
746
747 //         // appending to signal matrix
748 //         this->signalMatrix = \
749 //             torch::cat({this->signalMatrix, zero_tensor}, 1);
750 //     }
751
752 //     // breaking the signal into frames
753 //     fBuffer2D(this->signalMatrix, frame_size);
754
755
756 //     // add some zeros to the end of frames to accomodate delaying of signals.
757 //     torch::Tensor zero_filled_tensor = \
758 //         torch::zeros({this->signalMatrix.size(0), \
759 //                       this->num_buffer_zeros_per_frame, \
760 //                       this->num_sensors}).to(DATATYPE);
761 //     this->signalMatrix = \
762 //         torch::cat({this->signalMatrix, \
763 //                     zero_filled_tensor}, 1);
764
765
766 //     // tiling it to ensure that it works for all range-angle combinations
767 //     int number_of_azimuths_to_image = this->azimuth_centers.numel();
768 //     this->signalMatrix = \
769 //         torch::tile(this->signalMatrix, \
770 //                     {number_of_azimuths_to_image, 1, 1});
771
772 //     // element-wise multiplying the signals to delay each of the frame accordingly
773 //     this->signalMatrix = torch::mul(this->signalMatrix, \
774 //                                     this->mulFFTMMatrix);
775
776 //     // summing up the signals
777 //     // this->signalMatrix = torch::sum(this->signalMatrix, \
778 //                                     2, \
779 //                                     true);
780 //     this->signalMatrix = torch::sum(this->signalMatrix, \
781 //                                     2, \
782 //                                     false);
783
784
785 //     // printing some stuff
786 //     if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming: this->azimuth_centers.numel() =
"<<this->azimuth_centers.numel() <<std::endl;
787 //     if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming: this->range_centers.numel() =
"<<this->range_centers.numel() <<std::endl;
788 //     if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming: total number =
"<<this->range_centers.numel() * this->azimuth_centers.numel() <<std::endl;
789 //     if (DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_beamforming: this->signalMatrix.sizes().vec() =
"<<this->signalMatrix.sizes().vec() <<std::endl;
790
791 //     // creating a tensor to store the final image
792 //     torch::Tensor finalImage = \
793 //         torch::zeros({this->frame_size * this->range_centers.numel(), \
794 //                       this->azimuth_centers.numel()}).to(torch::kComplexFloat);

```

```

795
796
797 //      // creating a loop to assign values
798 //      for(int range_index = 0; range_index < this->range_centers.numel(); ++range_index){
799 //          for(int angle_index = 0; angle_index < this->azimuth_centers.numel(); ++angle_index){
800
801 //              // getting row index
802 //              int rowindex = \
803 //                  angle_index * this->range_centers.numel() \
804 //                  + range_index;
805
806 //              // getting the strip to store
807 //              torch::Tensor strip = \
808 //                  this->signalMatrix.index({rowindex, \
809 //                      torch::indexing::Slice()});
810
811 //              // taking just the first few values
812 //              strip = strip.index({torch::indexing::Slice(0, this->frame_size)});
813
814 //              // placing the strips on the image
815 //              finalImage.index_put_({\
816 //                  torch::indexing::Slice((range_index)*this->frame_size, \
817 //                      (range_index+1)*this->frame_size), \
818 //                  angle_index}, \
819 //                  strip);
820
821 //          }
822 //      }
823
824 //      // saving the image
825 //      this->beamformedImage = finalImage;
826
827
828
829 //      // converting image from polar to cartesian
830 //      nfdc_PolarToCartesian();
831
832
833 //  }
834
835 //  /* =====
836 //  Aim: Converting Polar Image to Cartesian
837 //  .....
838 //  Note:
839 //      > For now, we're assuming that the r value is one.
840 //  ----- */
841 //  void nfdc_PolarToCartesian(){
842
843
844 //      // deciding image dimensions
845 //      int num_pixels_width = 128;
846 //      int num_pixels_height = 128;
847
848
849 //      // creating query points
850 //      torch::Tensor max_right = \
851 //          torch::cos(\
852 //              torch::max(\
853 //                  this->azimuth_centers \
854 //                  - torch::mean(this->azimuth_centers) \
855 //                  + torch::tensor({90}).to(DATATYPE)) \
856 //              * PI/180);
857 //      torch::Tensor max_left = \
858 //          torch::cos(\
859 //              torch::min(this->azimuth_centers \
860 //                  - torch::mean(this->azimuth_centers) \
861 //                  + torch::tensor({90}).to(DATATYPE)) \
862 //              * PI/180);
863 //      torch::Tensor max_top = torch::tensor({1});
864 //      torch::Tensor max_bottom = torch::min(this->range_centers);
865
866
867 //      // creating query points along the x-dimension
868 //      torch::Tensor query_x = \
869 //          torch::linspace(

```

```

870 //          max_left,          \
871 //          max_right,         \
872 //          num_pixels_width   \
873 //          ).to(DATATYPE);
874
875 //      torch::Tensor query_y =          \
876 //      torch::linspace(                \
877 //          max_bottom.item<float>(),    \
878 //          max_top.item<float>(),      \
879 //          num_pixels_height          \
880 //          ).to(DATATYPE);
881
882
883 //      // converting original coordinates to their corresponding cartesian
884 //      float delta_r = 1/static_cast<float>(this->beamformedImage.size(0));
885 //      float delta_azimuth =          \
886 //      torch::abs(                    \
887 //          this->azimuth_centers.index({1}) \
888 //          - this->azimuth_centers.index({0}) \
889 //          ).item<float>();
890
891
892
893 //      // getting query points
894 //      torch::Tensor range_values = \
895 //      torch::linspace(            \
896 //          delta_r,                \
897 //          this->beamformedImage.size(0) * delta_r, \
898 //          this->beamformedImage.size(0)          \
899 //          ).to(DATATYPE);
900 //      range_values = \
901 //      range_values.reshape({range_values.numel(), 1});
902 //      range_values = \
903 //      torch::tile(range_values, \
904 //          {1, this->azimuth_centers.numel()});
905
906
907 //      // getting angle-values
908 //      torch::Tensor angle_values =          \
909 //      this->azimuth_centers                  \
910 //      - torch::mean(this->azimuth_centers)   \
911 //      + torch::tensor({90});
912 //      angle_values =          \
913 //      torch::tile(            \
914 //          angle_values,       \
915 //          {this->beamformedImage.size(0), 1});
916
917
918 //      // converting to cartesian original points
919 //      torch::Tensor query_original_x = \
920 //      range_values * torch::cos(angle_values * PI/180);
921 //      torch::Tensor query_original_y = \
922 //      range_values * torch::sin(angle_values * PI/180);
923
924
925
926 //      // converting points to vector 2D format
927 //      torch::Tensor query_source =          \
928 //      torch::cat({                          \
929 //          query_original_x.reshape({1, query_original_x.numel()}), \
930 //          query_original_y.reshape({1, query_original_y.numel()}), \
931 //          0;
932
933
934 //      // converting reflectivity to corresponding 2D format
935 //      torch::Tensor reflectivity_vectors = \
936 //      this->beamformedImage.reshape({1, this->beamformedImage.numel()});
937
938
939 //      // creating image
940 //      torch::Tensor cartesianImageLocal =          \
941 //      torch::zeros(                                \
942 //          {num_pixels_height,                      \
943 //          num_pixels_width}                        \
944 //          ).to(torch::kComplexFloat);

```

```

945
946 //      /*
947 //      Next Aim: start interpolating the points on the uniform grid.
948 //      */
949 //      #pragma omp parallel for
950 //      for(int x_index = 0; x_index < query_x.numel(); ++x_index){
951 //          // if(DEBUG_ULA) std::cout << "\t\t\t x_index = " << x_index << " ";
952 //          #pragma omp parallel for
953 //          for(int y_index = 0; y_index < query_y.numel(); ++y_index){
954 //              // if(DEBUG_ULA) if(y_index%16 == 0) std::cout<<". ";
955 //
956 //              // getting current values
957 //              torch::Tensor current_x = query_x.index({x_index}).reshape({1, 1});
958 //              torch::Tensor current_y = query_y.index({y_index}).reshape({1, 1});
959 //
960 //
961 //
962 //
963 //              // getting the query value
964 //              torch::Tensor query_vector = torch::cat({current_x, current_y}, 0);
965 //
966 //
967 //              // copying the query source
968 //              torch::Tensor query_source_relative = query_source;
969 //              query_source_relative = query_source_relative - query_vector;
970 //
971 //
972 //              // subsetting using absolute values and masks
973 //              float threshold = delta_r * 10;
974 //              // PRINTDOTS
975 //              auto mask_row = \
976 //                  torch::abs(query_source_relative[0]) <= threshold;
977 //              auto mask_col = \
978 //                  torch::abs(query_source_relative[1]) <= threshold;
979 //              auto mask_together = torch::mul(mask_row, mask_col);
980 //
981 //
982 //
983 //
984 //              // calculating number of points in threshold neighbourhood
985 //              int num_points_in_threshold_neighbourhood = \
986 //                  torch::sum(mask_together).item<int>();
987 //              if (num_points_in_threshold_neighbourhood == 0){
988 //                  continue;
989 //              }
990 //
991 //
992 //
993 //              // subsetting points in neighbourhood
994 //              torch::Tensor PointsInNeighbourhood = \
995 //                  query_source_relative.index({
996 //                      torch::indexing::Slice(), \
997 //                      mask_together});
998 //              torch::Tensor ReflectivitiesInNeighbourhood = \
999 //                  reflectivity_vectors.index({torch::indexing::Slice(), mask_together});
1000 //
1001 //
1002 //              // finding the distance between the points
1003 //              torch::Tensor relativeDistances = \
1004 //                  torch::linalg_norm(PointsInNeighbourhood, \
1005 //                      2, 0, true, \
1006 //                      torch::kFloat).to(DATATYPE);
1007 //
1008 //
1009 //              // calculating weighing factor
1010 //              torch::Tensor weighingFactor = \
1011 //                  torch::nn::functional::softmax( \
1012 //                      torch::max(relativeDistances)- relativeDistances, \
1013 //                      torch::nn::functional::SoftmaxFuncOptions(1));
1014 //
1015 //
1016 //              // combining intensities using distances
1017 //              torch::Tensor finalIntensity = \
1018 //                  torch::sum(
1019 //                      torch::mul(weighingFactor, \

```

```

1020 //                      ReflectivitiesInNeighbourhood));
1021
1022 //          // assigning values
1023 //          cartesianImageLocal.index_put_({x_index, y_index}, finalIntensity);
1024
1025 //      }
1026 //      // std::cout<<std::endl;
1027 //  }
1028
1029 //      // saving to member function
1030 //      this->cartesianImage = cartesianImageLocal;
1031
1032 //  }
1033
1034 //  /* =====
1035 //  Aim: create acoustic image directly
1036 //  ----- */
1037 //  void nfdc_createAcousticImage(ScattererClass* scatterers, \
1038 //                               TransmitterClass* transmitterObj){
1039
1040 //      // first we ensure that the scatterers are in our frame of reference
1041 //      scatterers->coordinates = scatterers->coordinates - this->location;
1042
1043 //      // finding the spherical coordinates of the scatterers
1044 //      torch::Tensor scatterers_spherical = fCart2Sph(scatterers->coordinates);
1045
1046 //      // note that its not precisely projection. its rotation. So the original lengths must be
1047 //      // maintained. but thats easy since the operation of putting the elevation to be zero works just fine.
1048 //      scatterers_spherical.index_put_({1, torch::indexing::Slice()}, 0);
1049
1050 //      // converting the points back to cartesian
1051 //      torch::Tensor scatterers_acoustic_cartesian = fSph2Cart(scatterers_spherical);
1052
1053 //      // removing the z-dimension
1054 //      scatterers_acoustic_cartesian = \
1055 //          scatterers_acoustic_cartesian.index({torch::indexing::Slice(0, 2, 1), \
1056 //                                              torch::indexing::Slice()});
1057
1058 //      // deciding image dimensions
1059 //      int num_pixels_x = 512;
1060 //      int num_pixels_y = 512;
1061 //      torch::Tensor acousticImage = \
1062 //          torch::zeros({num_pixels_x, \
1063 //                      num_pixels_y}).to(DATATYPE);
1064
1065 //      // finding the max and min values
1066 //      torch::Tensor min_x = torch::min(scatterers_acoustic_cartesian[0]);
1067 //      torch::Tensor max_x = torch::max(scatterers_acoustic_cartesian[0]);
1068 //      torch::Tensor min_y = torch::min(scatterers_acoustic_cartesian[1]);
1069 //      torch::Tensor max_y = torch::max(scatterers_acoustic_cartesian[1]);
1070
1071 //      // creating query grids
1072 //      torch::Tensor query_x = torch::linspace(0, 1, num_pixels_x);
1073 //      torch::Tensor query_y = torch::linspace(0, 1, num_pixels_y);
1074
1075 //      // scaling it up to image max-point spread
1076 //      query_x = min_x + (max_x - min_x) * query_x;
1077 //      query_y = min_y + (max_y - min_y) * query_y;
1078 //      float delta_queryx = (query_x[1] - query_x[0]).item<float>();
1079 //      float delta_queryy = (query_y[1] - query_y[0]).item<float>();
1080
1081 //      // creating a mesh-grid
1082 //      auto queryMeshGrid = torch::meshgrid({query_x, query_y}, "ij");
1083 //      query_x = queryMeshGrid[0].reshape({1, queryMeshGrid[0].numel()});
1084 //      query_y = queryMeshGrid[1].reshape({1, queryMeshGrid[1].numel()});
1085 //      torch::Tensor queryMatrix = torch::cat({query_x, query_y}, 0);
1086
1087 //      // printing shapes
1088 //      if(DEBUG_ULA) PRINTSMALLLINE
1089 //      if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: query_x.shape =
1090 //      "<<query_x.sizes().vec()<<std::endl;
1091 //      if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: query_y.shape =
1092 //      "<<query_y.sizes().vec()<<std::endl;
1093 //      if(DEBUG_ULA) std::cout<<"\t\t ULAClass::nfdc_createAcousticImage: queryMatrix.shape =
1094 //      "<<queryMatrix.sizes().vec()<<std::endl;

```



```

1091
1092 //      // setting up threshold values
1093 //      float threshold_value = \
1094 //          std::min(delta_queryx, \
1095 //              delta_queryy); if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage:
line 711"<<std::endl;
1096
1097 //      // putting a loop through the whole thing
1098 //      for(int i = 0; i<queryMatrix[0].numel(); ++i){
1099 //          // for each element in the query matrix
1100 //          if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 716"<<std::endl;
1101
1102 //          // calculating relative position of all the points
1103 //          torch::Tensor relativeCoordinates = \
1104 //              scatterers_acoustic_cartesian - \
1105 //              queryMatrix.index({torch::indexing::Slice(), i}).reshape({2, 1}); if(DEBUG_ULA)
std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 720"<<std::endl;
1106
1107 //          // calculating distances between all the points and the query point
1108 //          torch::Tensor relativeDistances = \
1109 //              torch::linalg_norm(relativeCoordinates, \
1110 //                  1, 0, true, \
1111 //                  DATATYPE);if(DEBUG_ULA) std::cout<<"\t\t\t
ULAClass::nfdc_createAcousticImage: line 727"<<std::endl;
1112 //          // finding points that are within the threshold
1113 //          torch::Tensor conditionMeetingPoints = \
1114 //              relativeDistances.squeeze() <= threshold_value;if(DEBUG_ULA) std::cout<<"\t\t\t
ULAClass::nfdc_createAcousticImage: line 729"<<std::endl;
1115
1116 //          // subsetting the points in the neighbourhood
1117 //          if(torch::sum(conditionMeetingPoints).item<float>() == 0){
1118
1119 //              // continuing implementation if there are no points in the neighbourhood
1120 //              continue; if(DEBUG_ULA) std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line
735"<<std::endl;
1121 //          }
1122 //          else{
1123 //              // creating mask for points in the neighbourhood
1124 //              auto mask = (conditionMeetingPoints == 1);if(DEBUG_ULA) std::cout<<"\t\t\t
ULAClass::nfdc_createAcousticImage: line 739"<<std::endl;
1125
1126 //              // subsetting relative distances in the neighbourhood
1127 //              torch::Tensor distanceInTheNeighbourhood = \
1128 //                  relativeDistances.index({torch::indexing::Slice(), mask});if(DEBUG_ULA)
std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 743"<<std::endl;
1129
1130 //              // subsetting reflectivity of points in the neighbourhood
1131 //              torch::Tensor reflectivityInTheNeighbourhood = \
1132 //                  scatterers->reflectivity.index({torch::indexing::Slice(), mask});if(DEBUG_ULA)
std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 747"<<std::endl;
1133
1134 //              // assigning intensity as a function of distance and reflectivity
1135 //              torch::Tensor reflectivityAssignment = \
1136 //                  torch::mul(torch::exp(-distanceInTheNeighbourhood), \
1137 //                      reflectivityInTheNeighbourhood);if(DEBUG_ULA) std::cout<<"\t\t\t
ULAClass::nfdc_createAcousticImage: line 752"<<std::endl;
1138 //              reflectivityAssignment = \
1139 //                  torch::sum(reflectivityAssignment);if(DEBUG_ULA) std::cout<<"\t\t\t
ULAClass::nfdc_createAcousticImage: line 754"<<std::endl;
1140
1141 //              // assigning this value to the image pixel intensity
1142 //              int pixel_position_x = i%num_pixels_x;
1143 //              int pixel_position_y = std::floor(i/num_pixels_x);
1144 //              acousticImage.index_put_({pixel_position_x, \
1145 //                  pixel_position_y}, \
1146 //                  reflectivityAssignment.item<float>());if(DEBUG_ULA)
std::cout<<"\t\t\t ULAClass::nfdc_createAcousticImage: line 761"<<std::endl;
1147 //          }
1148
1149 //      }
1150
1151 //      // storing the acoustic-image to the member
1152 //      this->currentArtificialAcousticImage = acousticImage;
1153
1154 //      // // saving the torch::tensor

```

```

1155 //      // torch::save(acousticImage, \
1156 //      //      "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/Assets/acoustic_image.pt");
1157
1158
1159
1160 //      // // bringing it back to the original coordinates
1161 //      // scatterers->coordinates = scatterers->coordinates + this->location;
1162 //  }
1163
1164
1165
1166 // };
1167
1168
1169 template <typename T>
1170 class ULAClass
1171 {
1172 public:
1173     // intrinsic parameters
1174     int          num_sensors;           // number of sensors
1175     T            inter_element_spacing; // space between sensors
1176     std::vector<std::vector<T>> coordinates; // coordinates of each sensor
1177     T            sampling_frequency;    // sampling frequency of the sensors
1178     T            recording_period;      // recording period of the ULA
1179     std::vector<T> location;            // location of first coordinate
1180
1181     // derived
1182     std::vector<T> sensor_direction;
1183     std::vector<std::vector<T>> signalMatrix;
1184
1185     // decimation related
1186     int          decimation_factor;      // the new decimation factor
1187     T            post_decimation_sampling_frequency; // the new sampling frequency
1188     std::vector<T> lowpass_filter_coefficients_for_decimation; // filter-coefficients for filtering
1189
1190     // imaging related
1191     T range_resolution; // theoretical range-resolution =  $\frac{c}{2B}$ 
1192     T azimuthal_resolution; // theoretical azimuth-resolution =  $\frac{\lambda}{(N-1) \cdot \text{inter-element-distance}}$ 
1193     T range_cell_size; // the range-cell quanta we're choosing for efficiency trade-off
1194     T azimuth_cell_size; // the azimuth quanta we're choosing
1195     std::vector<T> azimuth_centers; // tensor containing the azimuth centers
1196     std::vector<T> range_centers; // tensor containing the range-centers
1197     int frame_size; // the frame-size corresponding to a range cell in a decimated signal
1198     matrix
1199
1200     std::vector<std::vector<complex<T>>> mulFFTMMatrix; // the matrix containing the delays for each-element
1201     // as a slot
1202     std::vector<complex<T>> matchFilter; // torch tensor containing the match-filter
1203     int num_buffer_zeros_per_frame; // number of zeros we're adding per frame to ensure
1204     // no-rotation
1205     std::vector<std::vector<T>> beamformedImage; // the beamformed image
1206     std::vector<std::vector<T>> cartesianImage; // the cartesian version of beamformed image
1207
1208     // Artificial acoustic-image related
1209     std::vector<std::vector<T>> currentArtificialAcousticImage; // acoustic image directly produced
1210
1211     // Basic Constructor
1212     ULAClass() = default;
1213
1214     // constructor
1215     ULAClass(const int num_sensors_arg,
1216             const auto inter_element_spacing_arg,
1217             const auto& coordinates_arg,
1218             const auto& sampling_frequency_arg,
1219             const auto& recording_period_arg,
1220             const auto& location_arg,
1221             const auto& signalMatrix_arg,
1222             const auto& lowpass_filter_coefficients_for_decimation_arg):
1223         num_sensors(num_sensors_arg),
1224         inter_element_spacing(inter_element_spacing_arg),
1225         coordinates(std::move(coordinates_arg)),
1226         sampling_frequency(sampling_frequency_arg),
1227         recording_period(recording_period_arg),

```

```

1226         location(std::move(location_arg)),
1227         signalMatrix(std::move(signalMatrix_arg)),
1228         lowpass_filter_coefficients_for_decimation(std::move(lowpass_filter_coefficients_for_decimation_arg))
1229     {
1230
1231         // calculating ULA direction
1232         sensor_direction = std::vector<T>{coordinates[1][0] - coordinates[0][0],
1233                                           coordinates[1][1] - coordinates[0][1],
1234                                           coordinates[1][2] - coordinates[0][2]};
1235
1236         // normalizing
1237         auto norm_value_temp {std::inner_product(sensor_direction.begin(), sensor_direction.end(),
1238                                                  sensor_direction.begin(),
1239                                                  0.00)};
1240
1241         // dividing
1242         if (norm_value_temp != 0) {sensor_direction = sensor_direction / norm_value_temp;}
1243     }
1244
1245     // deleting copy constructor/assignment
1246     // ULAClass(const ULAClass& other)           = delete;
1247     // ULAClass& operator=(const ULAClass& other) = delete;
1248
1249
1250     // build sensor-coordinates based on location
1251     void buildCoordinatesBasedOnLocation();
1252
1253     /* =====
1254     Aim: Init
1255     ----- */
1256     void init(TransmitterClass<T>& transmitterObj);
1257
1258     /* =====
1259     Aim: Creating match-filter
1260     ----- */
1261     void nfdc_CreateMatchFilter(TransmitterClass<T>& transmitterObj);
1262
1263 };
1264
1265 // =====
1266 template <typename T>
1267 void ULAClass<T>::buildCoordinatesBasedOnLocation()
1268 {
1269
1270     // length-normalizing sensor-direction
1271     this->sensor_direction = this->sensor_direction / norm(this->sensor_direction);
1272
1273     // multiply with inter-element distance
1274     this->sensor_direction = this->sensor_direction * this->inter_element_spacing;
1275
1276     // create integer array
1277     auto integer_array {linspace<T>(0, this->num_sensors-1, this->num_sensors)};
1278     auto test {svr::tile(integer_array, {3,1}) * \
1279               svr::tile(transpose(this->sensor_direction),
1280                         {1, static_cast<std::size_t>(this->num_sensors)})};
1281     this->coordinates = this->location + test;
1282 }
1283
1284 /* =====
1285 Aim: Init
1286 ----- */
1287 template <typename T>
1288 void ULAClass<T>::init(TransmitterClass<T>& transmitterObj)
1289 {
1290
1291     // calculating range-related parameters
1292     this->range_resolution = 1500.00/(2 * transmitterObj.fc);
1293     this->range_cell_size = 40 * this->range_resolution;
1294
1295     // calculating azimuth-related parameters
1296     this->azimuthal_resolution = (1500.00 / transmitterObj.fc) / \
1297                                (this->num_sensors - 1) * (this->inter_element_spacing);
1298     this->azimuth_cell_size = 2 * this->azimuthal_resolution;
1299
1300     // creating and storing match-filter

```

```

1301     this->nfdc_CreateMatchFilter(transmitterObj);
1302 }
1303 }
1304
1305 /* =====
1306 Aim: Creating match-filter
1307 ----- */
1308 template <typename T>
1309 void ULAClass<T>::nfdc_CreateMatchFilter(TransmitterClass<T>& transmitterObj)
1310 {
1311     svr::Timer timer("nfdc_CreateMatchFilter");
1312
1313     // creating matrix for basebanding signal
1314     auto linspace00 {linspace(0,
1315                             transmitterObj.Signal.size()-1,
1316                             transmitterObj.Signal.size())};
1317     auto basebanding_vector {linspace00 * \
1318                             exp(-1.00 * 1i * 2.00 * std::numbers::pi * \
1319                                 (transmitterObj.fc / this->sampling_frequency)*\
1320                                 linspace00)};
1321
1322     // multiplying signal with basebanding signal
1323     auto match_filter {transmitterObj.Signal * basebanding_vector};
1324
1325     // low-pass filtering with baseband signal to obtain baseband signal
1326     match_filter = svr::conv1D(match_filter,
1327                               this->lowpass_filter_coefficients_for_decimation);
1328
1329     // creating sampling-indices
1330     int decimation_factor {static_cast<int>(std::floor(
1331         (static_cast<T>(this->sampling_frequency)/2.00) / \
1332         (static_cast<T>(transmitterObj.bandwidth)/2.00))
1333     });
1334     int final_num_samples {static_cast<int>(std::ceil(
1335         static_cast<T>(match_filter.size())/ \
1336         static_cast<T>(decimation_factor)
1337     ))};
1338     auto sampling_indices {
1339         linspace(1,
1340                 (final_num_samples - 1) * decimation_factor,
1341                 final_num_samples)
1342     };
1343
1344     // sampling the signal
1345     match_filter = svr::index(match_filter, sampling_indices);
1346
1347     // taking conjugate and flipping the signal
1348     match_filter = svr::fliplr( match_filter);
1349     match_filter = svr::conj( match_filter);
1350
1351     // storing the match-filter to the class member
1352     this->matchFilter = std::move(match_filter);
1353 }

```

```

57 //      //      // saving reflectivities
58 //      //      save(scatterer_fls.reflectivity, \
59 //      //      "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_fls_coordinates_reflectivity.pt");
60 //      //      save(scatterer_port.reflectivity, \
61 //      //      "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_port_coordinates_reflectivity.pt");
62 //      //      save(scatterer_starboard.reflectivity, \
63 //      //      "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_starboard.coordinates_reflectivity.pt");
64 //      //      "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/SeafloorScatter_starboard.coordinates_reflectivity.pt");
65
66 //      //      // plotting tensors
67 //      //      fPlotTensors();
68
69 //      //      // indicating end of thread
70 //      //      cout<<"\t\t\t\t\t\t\t Ended (timeID: "<<timeID<<)"<<endl;
71 //      //      }
72 // }
73
74
75 // // hash-defines
76 // #define PI                                3.14159265
77 // #define DEBUGMODE_AUV                      false
78 // #define SAVE_SIGNAL_MATRIX                 true
79 // #define SAVE_DECIMATED_SIGNAL_MATRIX       true
80 // #define SAVE_MATCHFILTERED_SIGNAL_MATRIX   true
81
82 // class AUVClass{
83 // public:
84 //     // Intrinsic attributes
85 //     Tensor location;                        // location of vessel
86 //     Tensor velocity;                       // current speed of the vessel [a vector]
87 //     Tensor acceleration;                   // current acceleration of vessel [a vector]
88 //     Tensor pointing_direction;             // direction to which the AUV is pointed
89
90 //     // uniform linear-arrays
91 //     ULAClass ULA_fls;                     // front-looking SONAR ULA
92 //     ULAClass ULA_port;                    // mounted ULA [object of class, ULAClass]
93 //     ULAClass ULA_starboard;               // mounted ULA [object of class, ULAClass]
94
95 //     // transmitters
96 //     TransmitterClass transmitter_fls;      // transmitter for front-looking SONAR
97 //     TransmitterClass transmitter_port;     // mounted transmitter [obj of class, TransmitterClass]
98 //     TransmitterClass transmitter_starboard; // mounted transmitter [obj of class, TransmitterClass]
99
100 //     // derived or dependent attributes
101 //     Tensor signalMatrix_1;                // matrix containing the signals obtained from ULA_1
102 //     Tensor largeSignalMatrix_1;           // matrix holding signal of synthetic aperture
103 //     Tensor beamformedLargeSignalMatrix;    // each column is the beamformed signal at each stop-hop
104
105 //     // plotting mode
106 //     bool plottingmode; // to suppress plotting associated with classes
107
108 //     // spotlight mode related
109 //     Tensor absolute_coords_patch_cart; // cartesian coordinates of patch
110
111 //     // Synthetic Aperture Related
112 //     Tensor ApertureSensorLocations; // sensor locations of aperture
113
114
115
116
117
118
119
120
121 // /* =====
122 // Aim: Init
123 // -----*/
124 // void init(){
125
126 //     // call sync-component attributes
127 //     this->syncComponentAttributes();
128 //     if (DEBUGMODE_AUV) cout << "AUVClass::init: line 128" << endl;

```

```

129
130 //      // initializing all the ULAs
131 //      this->ULA_fls.init(      &this->transmitter_fls);
132 //      this->ULA_port.init(      &this->transmitter_port);
133 //      this->ULA_starboard.init( &this->transmitter_starboard);
134 //      if (DEBUGMODE_AUV) cout << "AUVClass::init: line 134" << endl;
135
136
137 //      // precomputing delay-matrices for the ULA-class
138 //      thread ULA_fls_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
139 //      //      &this->ULA_fls, \
140 //      //      &this->transmitter_fls);
141 //      thread ULA_port_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
142 //      //      &this->ULA_port, \
143 //      //      &this->transmitter_port);
144 //      thread ULA_starboard_precompute_weights_t(&ULAClass::nfdc_precomputeDelayMatrices, \
145 //      //      &this->ULA_starboard, \
146 //      //      &this->transmitter_starboard);
147 //      if (DEBUGMODE_AUV) cout << "AUVClass::init: line 145" << endl;
148
149 //      // joining the threads back
150 //      ULA_fls_precompute_weights_t.join();
151 //      ULA_port_precompute_weights_t.join();
152 //      ULA_starboard_precompute_weights_t.join();
153
154 //  }
155
156
157
158 //  /*=====
159 //  Aim: stepping motion
160 //  -----*/
161 //  void step(float timestep){
162
163 //      // updating location
164 //      this->location = this->location + this->velocity * timestep;
165 //      if(DEBUGMODE_AUV) cout<<"\t AUVClass: page 81 \n";
166
167 //      // updating attributes of members
168 //      this->syncComponentAttributes();
169 //      if(DEBUGMODE_AUV) cout<<"\t AUVClass: page 85 \n";
170 //  }
171
172
173
174 //  /*=====
175 //  Aim: updateAttributes
176 //  -----*/
177 //  void syncComponentAttributes(){
178
179 //      // updating ULA attributes
180 //      if(DEBUGMODE_AUV) cout<<"\t AUVClass: page 97 \n";
181
182 //      // updating locations
183 //      this->ULA_fls.location      = this->location;
184 //      this->ULA_port.location     = this->location;
185 //      this->ULA_starboard.location = this->location;
186
187 //      // updating the pointing direction of the ULAs
188 //      Tensor ula_fls_sensor_direction_spherical = \
189 //      //      fCart2Sph(this->pointing_direction);      // spherical coords
190 //      //      ula_fls_sensor_direction_spherical[0]      = \
191 //      //      ula_fls_sensor_direction_spherical[0] - 90;
192 //      //      Tensor ula_fls_sensor_direction_cart      = \
193 //      //      fSph2Cart(ula_fls_sensor_direction_spherical);
194
195 //      this->ULA_fls.sensorDirection      = ula_fls_sensor_direction_cart; // assigning sensor directionf
196 //      this->ULA_port.sensorDirection     = -this->pointing_direction;      // assigning sensor direction
197 //      this->ULA_starboard.sensorDirection = -this->pointing_direction;      // assigning sensor direction
198 //      for ULA-Starboard
199 //      // // calling the function to update the arguments
200 //      // this->ULA_fls.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) cout<<"\t AUVClass: line 109

```

```

201 // \n";
202 // // this->ULA_port.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) cout<<"\t AUVClass: line 111
203 // \n";
204 // // this->ULA_starboard.buildCoordinatesBasedOnLocation(); if(DEBUGMODE_AUV) cout<<"\t AUVClass:
205 // line 113 \n";
206
207 // // updating transmitter locations
208 // this->transmitter_fls.location = this->location;
209 // this->transmitter_port.location = this->location;
210 // this->transmitter_starboard.location = this->location;
211
212 // // updating transmitter pointing directions
213 // this->transmitter_fls.updatePointingAngle( this->pointing_direction);
214 // this->transmitter_port.updatePointingAngle( this->pointing_direction);
215 // this->transmitter_starboard.updatePointingAngle( this->pointing_direction);
216 // }
217
218 // // =====
219 // Aim: operator overriding for printing
220 // -----*/
221 // friend ostream& operator<<(ostream& os, AUVClass &auv){
222 //     os<<"\t location = "<<transpose(auv.location, 0, 1)<<endl;
223 //     os<<"\t velocity = "<<transpose(auv.velocity, 0, 1)<<endl;
224 //     return os;
225 // }
226
227 // // =====
228 // Aim: Subsetting Scatterers
229 // -----*/
230 // void subsetScatterers(ScattererClass* scatterers,\
231 //                       TransmitterClass* transmitterObj,\
232 //                       float tilt_angle){
233 //
234 //     // ensuring components are synced
235 //     this->syncComponentAttributes();
236 //     if(DEBUGMODE_AUV) cout<<"\t AUVClass: page 120 \n";
237 //
238 //     // calling the method associated with the transmitter
239 //     if(DEBUGMODE_AUV) {cout<<"\t\t scatterers.shape = "; fPrintTensorSize(scatterers->coordinates);}
240 //     if(DEBUGMODE_AUV) cout<<"\t\t tilt_angle = "<<tilt_angle<<endl;
241 //     transmitterObj->subsetScatterers(scatterers, tilt_angle);
242 //     if(DEBUGMODE_AUV) cout<<"\t AUVClass: page 124 \n";
243 // }
244
245 // // yaw-correction matrix
246 // Tensor createYawCorrectionMatrix(Tensor pointing_direction_spherical, \
247 //                                  float target_azimuth_deg){
248 //
249 //     // building parameters
250 //     Tensor azimuth_correction = tensor({target_azimuth_deg}).to(DATATYPE).to(DEVICE) - \
251 //                                 pointing_direction_spherical[0];
252 //     Tensor azimuth_correction_radians = azimuth_correction * PI / 180;
253 //
254 //     Tensor yawCorrectionMatrix = \
255 //         tensor({cos(azimuth_correction_radians).item<float>(), \
256 //               cos(tensor({90}).to(DATATYPE).to(DEVICE)*PI/180 +
257 // azimuth_correction_radians).item<float>(), \
258 //               (float)0, \
259 //               sin(azimuth_correction_radians).item<float>(), \
260 //               sin(tensor({90}).to(DATATYPE).to(DEVICE)*PI/180 +
261 // azimuth_correction_radians).item<float>(), \
262 //               (float)0, \
263 //               (float)0, \
264 //               (float)1}).reshape({3,3}).to(DATATYPE).to(DEVICE);
265 //
266 //     // returning the matrix
267 //     return yawCorrectionMatrix;
268 // }
269
270 // // pitch-correction matrix
271 // Tensor createPitchCorrectionMatrix(Tensor pointing_direction_spherical, \

```



```

271 // float target_elevation_deg){
272 //
273 // // building parameters
274 // Tensor elevation_correction = tensor({target_elevation_deg}).to(DATATYPE).to(DEVICE) - \
275 // pointing_direction_spherical[1];
276 // Tensor elevation_correction_radians = elevation_correction * PI / 180;
277 //
278 // // creating the matrix
279 // Tensor pitchCorrectionMatrix = \
280 // tensor({(float)1, \
281 // (float)0, \
282 // (float)0, \
283 // (float)0, \
284 // cos(elevation_correction_radians).item<float>(), \
285 // cos(tensor({90}).to(DATATYPE).to(DEVICE)*PI/180 +
elevation_correction_radians).item<float>(),\
286 // (float)0, \
287 // sin(elevation_correction_radians).item<float>(), \
288 // sin(tensor({90}).to(DATATYPE).to(DEVICE)*PI/180 +
elevation_correction_radians).item<float>())}.reshape({3,3}).to(DATATYPE);
289 //
290 // // returning the matrix
291 // return pitchCorrectionMatrix;
292 // }
293 //
294 // // Signal Simulation
295 // void simulateSignal(ScattererClass& scatterer){
296 //
297 // // printing status
298 // cout << "\t AUVClass::simulateSignal: Began Signal Simulation" << endl;
299 //
300 // // making three copies
301 // ScattererClass scatterer_fls = scatterer;
302 // ScattererClass scatterer_port = scatterer;
303 // ScattererClass scatterer_starboard = scatterer;
304 //
305 // // finding the pointing direction in spherical
306 // Tensor auv_pointing_direction_spherical = fCart2Sph(this->pointing_direction);
307 //
308 // // asking the transmitters to subset the scatterers by multithreading
309 // thread transmitterFLSSubset_t(&AUVClass::subsetScatterers, this, \
310 // &scatterer_fls,\
311 // &this->transmitter_fls, \
312 // (float)0);
313 // thread transmitterPortSubset_t(&AUVClass::subsetScatterers, this, \
314 // &scatterer_port,\
315 // &this->transmitter_port, \
316 // auv_pointing_direction_spherical[1].item<float>());
317 // thread transmitterStarboardSubset_t(&AUVClass::subsetScatterers, this, \
318 // &scatterer_starboard, \
319 // &this->transmitter_starboard, \
320 // - auv_pointing_direction_spherical[1].item<float>());
321 //
322 // // joining the subset threads back
323 // transmitterFLSSubset_t.join();
324 // transmitterPortSubset_t.join();
325 // transmitterStarboardSubset_t.join();
326 //
327 //
328 // // multithreading the saving tensors part.
329 // thread savetensor_t(fSaveSeafloorScatteres, \
330 // scatterer, \
331 // scatterer_fls, \
332 // scatterer_port, \
333 // scatterer_starboard);
334 //
335 //
336 // // asking ULAs to simulate signal through multithreading
337 // thread ulaflds_signalsim_t(&ULAClass::nfdc_simulateSignal, \
338 // &this->ULA_fls, \
339 // &scatterer_fls, \
340 // &this->transmitter_fls);
341 // thread ulaport_signalsim_t(&ULAClass::nfdc_simulateSignal, \
342 // &this->ULA_port, \
343 // &scatterer_port, \

```

```

344 //                                     &this->transmitter_port);
345 //     thread ulastarboard_signalsim_t(&ULAClass::nfdc_simulateSignal, \
346 //                                     &this->ULA_starboard,          \
347 //                                     &scatterer_starboard,        \
348 //                                     &this->transmitter_starboard);
349
350 //     // joining them back
351 //     ulafls_signalsim_t.join();      // joining back the thread for ULA-FLS
352 //     ulaport_signalsim_t.join();     // joining back the signals-sim thread for ULA-Port
353 //     ulastarboard_signalsim_t.join(); // joining back the signal-sim thread for ULA-Starboard
354 //     savetensor_t.join();           // joining back the signal-sim thread for tensor-saving
355
356
357 // }
358
359 // // Imaging Function
360 // /* =====
361 // ----- */
362 // void image(){
363
364 //     // asking ULAs to decimate the signals obtained at each time step
365 //     thread ULA_fls_image_t(&ULAClass::nfdc_decimateSignal,      \
366 //                           &this->ULA_fls,                      \
367 //                           &this->transmitter_fls);
368 //     thread ULA_port_image_t(&ULAClass::nfdc_decimateSignal,    \
369 //                             &this->ULA_port,                   \
370 //                             &this->transmitter_port);
371 //     thread ULA_starboard_image_t(&ULAClass::nfdc_decimateSignal, \
372 //                                  &this->ULA_starboard,          \
373 //                                  &this->transmitter_starboard);
374
375 //     // joining the threads back
376 //     ULA_fls_image_t.join();
377 //     ULA_port_image_t.join();
378 //     ULA_starboard_image_t.join();
379
380 //     // saving the decimated signal
381 //     if (SAVE_DECIMATED_SIGNAL_MATRIX) {
382 //         cout << "\t AUVClass::image: saving decimated signal matrix" \
383 //              << endl;
384 //         save(this->ULA_fls.signalMatrix, \
385 //             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/decimated_signalMatrix_fls.pt");
386 //         save(this->ULA_port.signalMatrix, \
387 //             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/decimated_signalMatrix_port.pt");
388 //         save(this->ULA_starboard.signalMatrix, \
389 //             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/decimated_signalMatrix_starboard.pt");
390 //     }
391
392 //     // asking ULAs to match-filter the signals
393 //     thread ULA_fls_matchfilter_t(      \
394 //         &ULAClass::nfdc_matchFilterDecimatedSignal, \
395 //         &this->ULA_fls);
396 //     thread ULA_port_matchfilter_t(      \
397 //         &ULAClass::nfdc_matchFilterDecimatedSignal, \
398 //         &this->ULA_port);
399 //     thread ULA_starboard_matchfilter_t( \
400 //         &ULAClass::nfdc_matchFilterDecimatedSignal, \
401 //         &this->ULA_starboard);
402
403 //     // joining the threads back
404 //     ULA_fls_matchfilter_t.join();
405 //     ULA_port_matchfilter_t.join();
406 //     ULA_starboard_matchfilter_t.join();
407
408
409 //     // saving the decimated signal
410 //     if (SAVE_MATCHFILTERED_SIGNAL_MATRIX) {
411 //         // saving the tensors
412 //         cout << "\t AUVClass::image: saving match-filtered signal matrix" \
413 //              << endl;
414 //         save(this->ULA_fls.signalMatrix, \

```

```

416 //
417 //     "/Users/vrsreeganesesh/Documents/GitHub/AUV/Code/C++/Assets/matchfiltered_signalMatrix_fls.pt");
418 //     save(this->ULA_port.signalMatrix, \
419 //
420 //     "/Users/vrsreeganesesh/Documents/GitHub/AUV/Code/C++/Assets/matchfiltered_signalMatrix_port.pt");
421 //     save(this->ULA_starboard.signalMatrix, \
422 //
423 //     "/Users/vrsreeganesesh/Documents/GitHub/AUV/Code/C++/Assets/matchfiltered_signalMatrix_starboard.pt");
424 //
425 //         // running python-script
426 //     }
427 //
428 //         // performing the beamforming
429 //         thread ULA_fls_beamforming_t(&ULAClass::nfdc_beamforming, \
430 //             &this->ULA_fls, \
431 //             &this->transmitter_fls);
432 //         thread ULA_port_beamforming_t(&ULAClass::nfdc_beamforming, \
433 //             &this->ULA_port, \
434 //             &this->transmitter_port);
435 //         thread ULA_starboard_beamforming_t(&ULAClass::nfdc_beamforming, \
436 //             &this->ULA_starboard, \
437 //             &this->transmitter_starboard);
438 //
439 //         // joining the filters back
440 //         ULA_fls_beamforming_t.join();
441 //         ULA_port_beamforming_t.join();
442 //         ULA_starboard_beamforming_t.join();
443 //
444 //     }
445 //
446 //
447 //
448 //
449 //     /* =====
450 //     Aim: directly create acoustic image
451 //     ----- */
452 //     void createAcousticImage(ScattererClass* scatterers){
453 //
454 //         // making three copies
455 //         ScattererClass scatterer_fls    = scatterers;
456 //         ScattererClass scatterer_port   = scatterers;
457 //         ScattererClass scatterer_starboard = scatterers;
458 //
459 //         // printing size of scatterers before subsetting
460 //         PRINTSMALLLINE
461 //         cout<< "\t > AUVClass::createAcousticImage: Beginning Scatterer Subsetting"<<endl;
462 //         cout<< "\t AUVClass::createAcousticImage: scatterer_fls.coordinates.shape (before) = ";
463 //         fPrintTensorSize(scatterer_fls.coordinates);
464 //         cout<< "\t AUVClass::createAcousticImage: scatterer_port.coordinates.shape (before) = ";
465 //         fPrintTensorSize(scatterer_port.coordinates);
466 //         cout<< "\t AUVClass::createAcousticImage: scatterer_starboard.coordinates.shape (before) = ";
467 //         fPrintTensorSize(scatterer_starboard.coordinates);
468 //
469 //         // finding the pointing direction in spherical
470 //         Tensor auv_pointing_direction_spherical = fCart2Sph(this->pointing_direction);
471 //
472 //         // asking the transmitters to subset the scatterers by multithreading
473 //         thread transmitterFLSSubset_t(&AUVClass::subsetScatterers, this, \
474 //             &scatterer_fls, \
475 //             &this->transmitter_fls, \
476 //             (float)0);
477 //         thread transmitterPortSubset_t(&AUVClass::subsetScatterers, this, \
478 //             &scatterer_port, \
479 //             &this->transmitter_port, \
480 //             auv_pointing_direction_spherical[1].item<float>());
481 //         thread transmitterStarboardSubset_t(&AUVClass::subsetScatterers, this, \
482 //             &scatterer_starboard, \
483 //             &this->transmitter_starboard, \
484 //             - auv_pointing_direction_spherical[1].item<float>());
485 //
486 //         // joining the subset threads back
487 //         transmitterFLSSubset_t.join( );
488 //         transmitterPortSubset_t.join( );
489 //         transmitterStarboardSubset_t.join( );
490 //
491 //     }
492 //
493 //
494 //
495 //
496 //
497 //
498 //
499 //
500 //
501 //
502 //
503 //
504 //
505 //
506 //
507 //
508 //
509 //
510 //
511 //
512 //
513 //
514 //
515 //
516 //
517 //
518 //
519 //
520 //
521 //
522 //
523 //
524 //
525 //
526 //
527 //
528 //
529 //
530 //
531 //
532 //
533 //
534 //
535 //
536 //
537 //
538 //
539 //
540 //
541 //
542 //
543 //
544 //
545 //
546 //
547 //
548 //
549 //
550 //
551 //
552 //
553 //
554 //
555 //
556 //
557 //
558 //
559 //
560 //
561 //
562 //
563 //
564 //
565 //
566 //
567 //
568 //
569 //
570 //
571 //
572 //
573 //
574 //
575 //
576 //
577 //
578 //
579 //
580 //
581 //
582 //
583 //
584 //
585 //
586 //
587 //
588 //
589 //
590 //
591 //
592 //
593 //
594 //
595 //
596 //
597 //
598 //
599 //
600 //
601 //
602 //
603 //
604 //
605 //
606 //
607 //
608 //
609 //
610 //
611 //
612 //
613 //
614 //
615 //
616 //
617 //
618 //
619 //
620 //
621 //
622 //
623 //
624 //
625 //
626 //
627 //
628 //
629 //
630 //
631 //
632 //
633 //
634 //
635 //
636 //
637 //
638 //
639 //
640 //
641 //
642 //
643 //
644 //
645 //
646 //
647 //
648 //
649 //
650 //
651 //
652 //
653 //
654 //
655 //
656 //
657 //
658 //
659 //
660 //
661 //
662 //
663 //
664 //
665 //
666 //
667 //
668 //
669 //
670 //
671 //
672 //
673 //
674 //
675 //
676 //
677 //
678 //
679 //
680 //
681 //
682 //
683 //
684 //
685 //
686 //
687 //
688 //
689 //
690 //
691 //
692 //
693 //
694 //
695 //
696 //
697 //
698 //
699 //
700 //
701 //
702 //
703 //
704 //
705 //
706 //
707 //
708 //
709 //
710 //
711 //
712 //
713 //
714 //
715 //
716 //
717 //
718 //
719 //
720 //
721 //
722 //
723 //
724 //
725 //
726 //
727 //
728 //
729 //
730 //
731 //
732 //
733 //
734 //
735 //
736 //
737 //
738 //
739 //
740 //
741 //
742 //
743 //
744 //
745 //
746 //
747 //
748 //
749 //
750 //
751 //
752 //
753 //
754 //
755 //
756 //
757 //
758 //
759 //
760 //
761 //
762 //
763 //
764 //
765 //
766 //
767 //
768 //
769 //
770 //
771 //
772 //
773 //
774 //
775 //
776 //
777 //
778 //
779 //
780 //
781 //
782 //
783 //
784 //
785 //
786 //
787 //
788 //
789 //
790 //
791 //
792 //
793 //
794 //
795 //
796 //
797 //
798 //
799 //
800 //
801 //
802 //
803 //
804 //
805 //
806 //
807 //
808 //
809 //
810 //
811 //
812 //
813 //
814 //
815 //
816 //
817 //
818 //
819 //
820 //
821 //
822 //
823 //
824 //
825 //
826 //
827 //
828 //
829 //
830 //
831 //
832 //
833 //
834 //
835 //
836 //
837 //
838 //
839 //
840 //
841 //
842 //
843 //
844 //
845 //
846 //
847 //
848 //
849 //
850 //
851 //
852 //
853 //
854 //
855 //
856 //
857 //
858 //
859 //
860 //
861 //
862 //
863 //
864 //
865 //
866 //
867 //
868 //
869 //
870 //
871 //
872 //
873 //
874 //
875 //
876 //
877 //
878 //
879 //
880 //
881 //
882 //
883 //
884 //
885 //
886 //
887 //
888 //
889 //
890 //
891 //
892 //
893 //
894 //
895 //
896 //
897 //
898 //
899 //
900 //
901 //
902 //
903 //
904 //
905 //
906 //
907 //
908 //
909 //
910 //
911 //
912 //
913 //
914 //
915 //
916 //
917 //
918 //
919 //
920 //
921 //
922 //
923 //
924 //
925 //
926 //
927 //
928 //
929 //
930 //
931 //
932 //
933 //
934 //
935 //
936 //
937 //
938 //
939 //
940 //
941 //
942 //
943 //
944 //
945 //
946 //
947 //
948 //
949 //
950 //
951 //
952 //
953 //
954 //
955 //
956 //
957 //
958 //
959 //
960 //
961 //
962 //
963 //
964 //
965 //
966 //
967 //
968 //
969 //
970 //
971 //
972 //
973 //
974 //
975 //
976 //
977 //
978 //
979 //
980 //
981 //
982 //
983 //
984 //
985 //
986 //
987 //
988 //
989 //
990 //
991 //
992 //
993 //
994 //
995 //
996 //
997 //
998 //
999 //
1000 //

```

```

485 //      transmitterPortSubset_t.join( );
486 //      transmitterStarboardSubset_t.join( );
487
488
489 //      // asking the ULAs to directly create acoustic images
490 //      thread ULA_fls_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, this->ULA_fls, \
491 //                                     &scatterer_fls, &this->transmitter_fls);
492 //      thread ULA_port_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, &this->ULA_port, \
493 //                                     &scatterer_port, &this->transmitter_port);
494 //      thread ULA_starboard_acoustic_image_t(&ULAClass::nfdc_createAcousticImage, &this->ULA_starboard, \
495 //                                     &scatterer_starboard, &this->transmitter_starboard);
496
497 //      // joining the threads back
498 //      ULA_fls_acoustic_image_t.join( );
499 //      ULA_port_acoustic_image_t.join( );
500 //      ULA_starboard_acoustic_image_t.join();
501
502 //  }
503
504
505 // };
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528 // // 0.0000,
529 // // 0.0000,
530 // // 0.0000,
531 // // 0.0000,
532 // // 0.0000,
533 // // 0.0000,
534 // // 0.0000,
535 // // 0.0000,
536 // // 0.0000,
537 // // 0.0000,
538 // // 0.0000,
539 // // 0.0000,
540 // // 0.0000,
541 // // 0.0000,
542 // // 0.0000,
543 // // 0.0000,
544 // // 0.0000,
545 // // 0.0000,
546 // // 0.0000,
547 // // 0.0000,
548 // // 0.0000,
549 // // 0.0000,
550 // // 0.0000,
551 // // 0.0000,
552 // // 0.0000,
553 // // 0.0000,
554 // // 0.0000,
555 // // 0.0000,
556 // // 0.0000,
557 // // 0.0000,
558 // // 0.0000,
559 // // 0.0001,

```

```

560 // // 0.0001,
561 // // 0.0002,
562 // // 0.0003,
563 // // 0.0006,
564 // // 0.0009,
565 // // 0.0014,
566 // // 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
567
568
569
570
571
572
573
574
575
576
577
578
579 template <typename T>
580 class AUVClass{
581 public:
582
583     // Intrinsic attributes
584     std::vector<T>    location;           // location of vessel
585     std::vector<T>    velocity;          // velocity of the vessel
586     std::vector<T>    acceleration;      // acceleration of vessel
587     std::vector<T>    pointing_direction; // AUV's pointing direction
588
589     // uniform linear-arrays
590     ULAClass<T>        ULA_fls;          // front-looking SONAR ULA
591     ULAClass<T>        ULA_portside;     // mounted ULA [object of class, ULAClass]
592     ULAClass<T>        ULA_starboard;    // mounted ULA [object of class, ULAClass]
593
594     // transmitters
595     TransmitterClass<T> transmitter_fls; // transmitter for front-looking SONAR
596     TransmitterClass<T> transmitter_portside; // mounted transmitter [obj of class, TransmitterClass]
597     TransmitterClass<T> transmitter_starboard; // mounted transmitter [obj of class, TransmitterClass]
598
599     // derived or dependent attributes
600     std::vector<std::vector<T>> signalMatrix_1; // matrix containing the signals obtained from
        ULA_1
601     std::vector<std::vector<T>> largeSignalMatrix_1; // matrix holding signal of synthetic aperture
602     std::vector<std::vector<T>> beamformedLargeSignalMatrix; // each column is the beamformed signal at each
        stop-hop
603
604     // plotting mode
605     bool plottingmode; // to suppress plotting associated with classes
606
607     // spotlight mode related
608     std::vector<std::vector<T>> absolute_coords_patch_cart; // cartesian coordinates of patch
609
610     // Synthetic Aperture Related
611     std::vector<std::vector<T>> ApertureSensorLocations; // sensor locations of aperture
612
613     // functions
614     void syncComponentAttributes();
615     void init();
616
617 };
618
619 /*=====
620 Aim: update attributes
621 -----*/
622 template <typename T>
623 void AUVClass<T>::syncComponentAttributes()
624 {
625     // updating locations of ULAs
626     this->ULA_fls.location = this->location;
627     this->ULA_portside.location = this->location;
628     this->ULA_starboard.location = this->location;
629
630

```

```

631 // updating pointing-direction of ULAs
632 auto ula_fls_sensor_direction_spherical {svr::cart2sph(this->pointing_direction)};
633 ula_fls_sensor_direction_spherical[0] -= 90;
634 auto ula_fls_sensor_direction_cart {svr::sph2cart(ula_fls_sensor_direction_spherical)};
635
636 this->ULA_fls.sensor_direction = ula_fls_sensor_direction_cart;
637 this->ULA_portside.sensor_direction = -1 * this->pointing_direction;
638 this->ULA_starboard.sensor_direction = -1 * this->pointing_direction;
639
640
641 // calling function to update argumentss
642 this->ULA_fls.buildCoordinatesBasedOnLocation();
643 this->ULA_portside.buildCoordinatesBasedOnLocation();
644 this->ULA_starboard.buildCoordinatesBasedOnLocation();
645
646 // updating transmitter location
647 this->transmitter_fls.location = this->location;
648 this->transmitter_portside.location = this->location;
649 this->transmitter_starboard.location = this->location;
650
651 // updating transmitter pointing direction
652 this->transmitter_fls.updatePointingAngle( this->pointing_direction);
653 this->transmitter_portside.updatePointingAngle( this->pointing_direction);
654 this->transmitter_starboard.updatePointingAngle( this->pointing_direction);
655
656 }
657
658 /* =====
659 Aim: Init
660 -----*/
661 template <typename T>
662 void AUVClass<T>::init()
663 {
664     // call sync-component attributes
665     this->syncComponentAttributes();
666
667     // initializing all the ULAs
668     this->ULA_fls.init( this->transmitter_fls);
669     this->ULA_portside.init( this->transmitter_portside);
670     this->ULA_starboard.init( this->transmitter_starboard);
671
672     // pre-computing delay-matrices for ULA-class
673
674 }

```

8.2 Setup Scripts

8.2.1 Seafloor Setup

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

```

1 void fSeaFloorSetup(ScattererClass<double>& scatterers){
2
3     // auto save_files {false};
4     const auto save_files {false};
5     const auto hill_creation_flag {true};
6
7     // sea-floor bounds
8     auto bed_width {100.00};
9     auto bed_length {100.00};
10
11     // creating tensors for coordinates and reflectivity
12     vector<vector<double>> box_coordinates;
13     vector<double> box_reflectivity;
14
15     // scatter density
16     auto bed_width_density {static_cast<double>( 10.00)};
17     auto bed_length_density {static_cast<double>( 10.00)};
18
19     // setting up coordinates
20     auto xpoints {linspace<double>(0.00,
21                                     bed_width,
22                                     bed_width * bed_width_density)};
23     auto ypoints {linspace<double>(0.00,
24                                     bed_length,
25                                     bed_length * bed_length_density)};
26     if(save_files) fWriteVector(xpoints, "../csv-files/xpoints.csv"); // verified
27     if(save_files) fWriteVector(ypoints, "../csv-files/ypoints.csv"); // verified
28
29     // creating mesh
30     auto [xgrid, ygrid] = meshgrid(std::move(xpoints), std::move(ypoints));
31     if(save_files) fWriteMatrix(xgrid, "../csv-files/xgrid.csv"); // verified
32     if(save_files) fWriteMatrix(ygrid, "../csv-files/ygrid.csv"); // verified
33
34     // reshaping
35     auto X {reshape(xgrid, xgrid.size()*xgrid[0].size())};
36     auto Y {reshape(ygrid, ygrid.size()*ygrid[0].size())};
37     if(save_files) fWriteVector(X, "../csv-files/X.csv"); // verified
38     if(save_files) fWriteVector(Y, "../csv-files/Y.csv"); // verified
39
40     // creating heights of scatterers
41     if(hill_creation_flag){
42
43         // setting up hill parameters
44         auto num_hills {10};
45
46         // setting up placement of hills
47         auto points2D {concatenate<0>(X, Y)}; // verified
48         auto min2D {min<1, double>(points2D)}; // verified
49         auto max2D {max<1, double>(points2D)}; // verified
50         auto hill_2D_center {min2D + \
51                               rand({2, num_hills}) * (max2D - min2D)}; // verified
52
53         // setup: hill-dimensions
54         auto hill_dimensions_min {transpose(vector<double>{5, 5, 2})}; // verified
55         auto hill_dimensions_max {transpose(vector<double>{30, 30, 10})}; // verified
56         auto hill_dimensions {hill_dimensions_min + \
57                               rand({3, num_hills}) * (hill_dimensions_max - hill_dimensions_min)};
58                               // verified
59
60         // function-call: hill-creation function
61         fCreateHills(hill_2D_center,
62                     hill_dimensions,
63                     points2D);
64
65         // setting up floor reflectivity
66         auto floorScatter_reflectivity {std::vector<double>(Y.size(), 1.00)};

```

```
66
67     // populating the values of the incoming argument
68     scatterers.coordinates = std::move(points2D);
69     scatterers.reflectivity = std::move(floorScatter_reflectivity);
70
71 }
72 else{
73
74     // assigning flat heights
75     auto Z      {std::vector<double>(Y.size(), 0)};
76
77     // setting up floor coordinates
78     auto floorScatter_coordinates {concatenate<0>(X, Y, Z)};
79     auto floorScatter_reflectivity {std::vector<double>(Y.size(), 1)};
80
81     // populating the values of the incoming argument
82     scatterers.coordinates = std::move(floorScatter_coordinates);
83     scatterers.reflectivity = std::move(floorScatter_reflectivity);
84
85 }
86 }
```

8.2.2 Transmitter Setup

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

```

1  template <typename T>
2  void fTransmitterSetup(TransmitterClass<T>& transmitter_fls,
3                        TransmitterClass<T>& transmitter_portside,
4                        TransmitterClass<T>& transmitter_starboard)
5  {
6      // Setting up transmitter
7      T    sampling_frequency    {160e3};           // sampling frequency
8      T    f1                    {50e3};           // first frequency of LFM
9      T    f2                    {70e3};           // second frequency of LFM
10     T    fc                    {(f1 + f2)/2.00};   // finding center-frequency
11     T    bandwidth              {std::abs(f2 - f1)}; // bandwidth
12     T    pulselength            {5e-2};           // time of recording
13
14     // building LFM
15     auto  timearray             {linspace<T>(0.00,
16                                           pulselength,
17                                           std::floor(pulselength * sampling_frequency))};
18     auto  K                     {f2 - f1/pulselength}; // calculating frequency-slope
19     auto  Signal                {cos(2 * std::numbers::pi * \
20                                   (f1 + K*timearray) * \
21                                   timearray)};        // frequency at each time-step, with f1 = 0
22
23     // Setting up transmitter
24     auto  location               {std::vector<T>(3, 0)}; // location of transmitter
25     T    azimuthal_angle_fls     {0};                // initial pointing direction
26     T    azimuthal_angle_port    {90};               // initial pointing direction
27     T    azimuthal_angle_starboard {-90};            // initial pointing direction
28
29     T    elevation_angle         {-60};              // initial pointing direction
30
31     T    azimuthal_beamwidth_fls {20};               // azimuthal beamwidth of the signal
32     cone azimuthal_beamwidth_port {20};              // azimuthal beamwidth of the signal
33     cone azimuthal_beamwidth_starboard {20};         // azimuthal beamwidth of the signal
34     cone
35     T    elevation_beamwidth_fls {20};               // elevation beamwidth of the signal
36     cone elevation_beamwidth_port {20};              // elevation beamwidth of the signal
37     cone elevation_beamwidth_starboard {20};         // elevation beamwidth of the signal
38     cone
39     int  azimuthQuantDensity      {10}; // number of points, a degree is split into quantization
40         density along azimuth (used for shadowing)
41     int  elevationQuantDensity    {10}; // number of points, a degree is split into quantization
42         density along elevation (used for shadowing)
43     T    rangeQuantSize           {10}; // the length of a cell (used for shadowing)
44
45     T    azimuthShadowThreshold   {1}; // azimuth threshold (in degrees)
46     T    elevationShadowThreshold {1}; // elevation threshold (in degrees)
47
48     // transmitter-fls
49     transmitter_fls.location      = location;        // Assigning location
50     transmitter_fls.Signal        = Signal;          // Assigning signal
51     transmitter_fls.azimuthal_angle = azimuthal_angle_fls; // assigning azimuth angle
52     transmitter_fls.elevation_angle = elevation_angle;    // assigning elevation angle
53     transmitter_fls.azimuthal_beamwidth = azimuthal_beamwidth_fls; // assigning azimuth-beamwidth
54     transmitter_fls.elevation_beamwidth = elevation_beamwidth_fls; // assigning elevation-beamwidth
55     // updating quantization densities
56     transmitter_fls.azimuthQuantDensity = azimuthQuantDensity; // assigning azimuth quant density
57     transmitter_fls.elevationQuantDensity = elevationQuantDensity; // assigning elevation quant density
58     transmitter_fls.rangeQuantSize = rangeQuantSize; // assigning range-quantization
59     transmitter_fls.azimuthShadowThreshold = azimuthShadowThreshold; // azimuth-threshold in shadowing
60     transmitter_fls.elevationShadowThreshold = elevationShadowThreshold; // elevation-threshold in shadowing
61     // signal related
62     transmitter_fls.f_low         = f1;              // assigning lower frequency
63     transmitter_fls.f_high        = f2;              // assigning higher frequency

```

```

63 transmitter_fls.fc = fc; // assigning center frequency
64 transmitter_fls.bandwidth = bandwidth; // assigning bandwidth
65
66
67 // transmitter-portside
68 transmitter_portside.location = location; // Assigning location
69 transmitter_portside.Signal = Signal; // Assigning signal
70 transmitter_portside.azimuthal_angle = azimuthal_angle_port; // assigning azimuth angle
71 transmitter_portside.elevation_angle = elevation_angle; // assigning elevation angle
72 transmitter_portside.azimuthal_beamwidth = azimuthal_beamwidth_port; // assigning azimuth-beamwidth
73 transmitter_portside.elevation_beamwidth = elevation_beamwidth_port; // assigning elevation-beamwidth
74 // updating quantization densities
75 transmitter_portside.azimuthQuantDensity = azimuthQuantDensity; // assigning azimuth quant density
76 transmitter_portside.elevationQuantDensity = elevationQuantDensity; // assigning elevation quant
    density
77 transmitter_portside.rangeQuantSize = rangeQuantSize; // assigning range-quantization
78 transmitter_portside.azimuthShadowThreshold = azimuthShadowThreshold; // azimuth-threshold in shadowing
79 transmitter_portside.elevationShadowThreshold = elevationShadowThreshold; // elevation-threshold in
    shadowing
80 // signal related
81 transmitter_portside.f_low = f1; // assigning lower frequency
82 transmitter_portside.f_high = f2; // assigning higher frequency
83 transmitter_portside.fc = fc; // assigning center frequency
84 transmitter_portside.bandwidth = bandwidth; // assigning bandwidth
85
86
87 // transmitter-starboard
88 transmitter_starboard.location = location; // assigning location
89 transmitter_starboard.Signal = Signal; // assigning signal
90 transmitter_starboard.azimuthal_angle = azimuthal_angle_starboard; // assigning azimuthal signal
91 transmitter_starboard.elevation_angle = elevation_angle;
92 transmitter_starboard.azimuthal_beamwidth = azimuthal_beamwidth_starboard;
93 transmitter_starboard.elevation_beamwidth = elevation_beamwidth_starboard;
94 // updating quantization densities
95 transmitter_starboard.azimuthQuantDensity = azimuthQuantDensity; // assigning
    azimuth-quant-density
96 transmitter_starboard.elevationQuantDensity = elevationQuantDensity;
97 transmitter_starboard.rangeQuantSize = rangeQuantSize;
98 transmitter_starboard.azimuthShadowThreshold = azimuthShadowThreshold;
99 transmitter_starboard.elevationShadowThreshold = elevationShadowThreshold;
100 // signal related
101 transmitter_starboard.f_low = f1; // assigning lower frequency
102 transmitter_starboard.f_high = f2; // assigning higher frequency
103 transmitter_starboard.fc = fc; // assigning center frequency
104 transmitter_starboard.bandwidth = bandwidth; // assigning bandwidth
105
106 }

```

8.2.3 ULA Setup

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

```

1  template <typename T>
2  void fULASetup(ULAClass<T>&  ula_fls,
3                 ULAClass<T>&   ula_portside,
4                 ULAClass<T>&   ula_starboard)
5  {
6      // setting up ula
7      auto    num_sensors          {static_cast<int>(64)};           // number of sensors
8      T        sampling_frequency  {static_cast<T>(160e3)};         // sampling frequency
9      T        inter_element_spacing {1500/(2*sampling_frequency)}; // space between samples
10     T        recording_period     {10e-2};                       // sampling-period
11
12     // building the direction for the sensors
13     auto    ULA_direction          {std::vector<T>({-1, 0, 0})};
14     auto    ULA_direction_norm     {norm(ULA_direction)};
15     if (ULA_direction_norm != 0)   {ULA_direction = ULA_direction/ULA_direction_norm;}
16     ULA_direction                  = ULA_direction * inter_element_spacing;
17
18     // building coordinates for sensors
19     auto    ULA_coordinates         {transpose(ULA_direction) * \
20                                     linspace<double>(0.00,
21                                                         num_sensors -1,
22                                                         num_sensors)};
23
24     // coefficients of decimation filter
25     auto    lowpassfiltercoefficients {std::vector<T>{0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0001,
26                                                         0.0003, 0.0006, 0.0015, 0.0030, 0.0057, 0.0100, 0.0163, 0.0251, 0.0364, 0.0501, 0.0654, 0.0814,
27                                                         0.0966, 0.1093, 0.1180, 0.1212, 0.1179, 0.1078, 0.0914, 0.0699, 0.0451, 0.0192, -0.0053, -0.0262,
28                                                         -0.0416, -0.0504, -0.0522, -0.0475, -0.0375, -0.0239, -0.0088, 0.0057, 0.0179, 0.0263, 0.0303,
29                                                         0.0298, 0.0253, 0.0177, 0.0086, -0.0008, -0.0091, -0.0153, -0.0187, -0.0191, -0.0168, -0.0123,
30                                                         -0.0065, -0.0004, 0.0052, 0.0095, 0.0119, 0.0125, 0.0112, 0.0084, 0.0046, 0.0006, -0.0031, -0.0060,
31                                                         -0.0078, -0.0082, -0.0075, -0.0057, -0.0033, -0.0006, 0.0019, 0.0039, 0.0051, 0.0055, 0.0050, 0.0039,
32                                                         0.0023, 0.0005, -0.0012, -0.0025, -0.0034, -0.0036, -0.0034, -0.0026, -0.0016, -0.0004, 0.0007,
33                                                         0.0016, 0.0022, 0.0024, 0.0023, 0.0018, 0.0011, 0.0003, -0.0004, -0.0011, -0.0015, -0.0016, -0.0015}};
34
35     // assigning values
36
37     ula_fls.num_sensors                = num_sensors;           // assigning number of sensors
38     ula_fls.inter_element_spacing      = inter_element_spacing; // assigning inter-element
39     spacing
40     ula_fls.coordinates                = ULA_coordinates;       // assigning ULA coordinates
41     ula_fls.sampling_frequency         = sampling_frequency;     // assigning sampling
42     frequencys
43     ula_fls.recording_period           = recording_period;       // assigning recording period
44     ula_fls.sensor_direction           = ULA_direction;         // ULA direction
45     ula_fls.lowpass_filter_coefficients_for_decimation = lowpassfiltercoefficients; // storing coefficients
46
47     // assigning values
48
49     ula_portside.num_sensors           = num_sensors;           // assigning number of
50     sensors
51     ula_portside.inter_element_spacing  = inter_element_spacing; // assigning inter-element
52     spacing
53     ula_portside.coordinates           = ULA_coordinates;       // assigning ULA
54     coordinates
55     ula_portside.sampling_frequency     = sampling_frequency;     // assigning sampling
56     frequencys
57     ula_portside.recording_period       = recording_period;       // assigning recording
58     period
59     ula_portside.sensor_direction       = ULA_direction;         // ULA direction
60     ula_portside.lowpass_filter_coefficients_for_decimation = lowpassfiltercoefficients; // storing
61     coefficients
62
63     // assigning values
64
65     ula_starboard.num_sensors          = num_sensors;           // assigning number of
66     sensors
67     ula_starboard.inter_element_spacing = inter_element_spacing; // assigning
68     inter-element spacing
69     ula_starboard.coordinates          = ULA_coordinates;       // assigning ULA
70     coordinates
71     ula_starboard.sampling_frequency   = sampling_frequency;     // assigning sampling
72     frequencys

```

```
52     frequencys
    ula_starboard.recording_period          = recording_period;          // assigning recording
    period
53     ula_starboard.sensor_direction        = ULA_direction;            // ULA direction
54     ula_starboard.lowpass_filter_coefficients_for_decimation = lowpassfiltercoefficients; // storing
    coefficients
55 }
```

8.2.4 AUV Setup

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

```

1  // /* =====
2  // Aim: Setup sea floor
3  // NOAA: 50 to 100 KHz is the transmission frequency
4  // we'll create our LFM with 50 to 70KHz
5  // =====*/
6
7  // #ifndef DEVICE
8  //     #define DEVICE      torch::kMPS
9  //     // #define DEVICE      torch::kCPU
10 // #endif
11
12 // // =====
13 // void AUVSetup(AUVClass* auv) {
14
15 //     // building properties for the auv
16 //     torch::Tensor location      = torch::tensor({0,50,30}).reshape({3,1}).to(DATATYPE).to(DEVICE); //
17 //     starting location of AUV
18 //     torch::Tensor velocity      = torch::tensor({5,0, 0}).reshape({3,1}).to(DATATYPE).to(DEVICE); //
19 //     starting velocity of AUV
20 //     torch::Tensor pointing_direction = torch::tensor({1,0, 0}).reshape({3,1}).to(DATATYPE).to(DEVICE); //
21 //     pointing direction of AUV
22
23 //     // assigning
24 //     auv->location      = location;          // assigning location of auv
25 //     auv->velocity      = velocity;          // assigning vector representing velocity
26 //     auv->pointing_direction = pointing_direction; // assigning pointing direction of auv
27 // }
28
29
30
31
32
33
34 // =====
35 template <typename T>
36 void fAUVSetup(AUVClass<T>& auv) {
37
38 //     // building properties for the auv
39 //     auto location      {std::vector<T>{0, 50, 30}}; // starting location of AUV
40 //     auto velocity      {std::vector<T>{5, 0, 0}}; // starting velocity of AUV
41 //     auto pointing_direction {std::vector<T>{1, 0, 0}}; // pointing direction of AUV
42
43 //     // assigning
44 //     auv.location      = std::move(location); // assigning location of auv
45 //     auv.velocity      = std::move(velocity); // assigning vector representing velocity
46 //     auv.pointing_direction = std::move(pointing_direction); // assigning pointing direction of auv
47
48 }

```

8.3 Function Definitions

8.3.1 Cartesian Coordinates to Spherical Coordinates

```

1  /* =====
2  Aim: Setup sea floor
3  =====*/
4  #include <torch/torch.h>
5  #include <iostream>
6
7  // hash-defines
8  #define PI          3.14159265
9  #define DEBUG_Cart2Sph false
10
11 #ifndef DEVICE
12     #define DEVICE      torch::kMPS
13     // #define DEVICE    torch::kCPU
14 #endif
15
16
17 // bringing in functions
18 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Functions/fPrintTensorSize.cpp"
19
20 #pragma once
21
22 torch::Tensor fCart2Sph(torch::Tensor cartesian_vector){
23
24     // sending argument to the device
25     cartesian_vector = cartesian_vector.to(DEVICE);
26     if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 26 \n";
27
28     // splatting the point onto xy plane
29     torch::Tensor xysplat = cartesian_vector.clone().to(DEVICE);
30     xysplat[2] = 0;
31     if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 31 \n";
32
33     // finding splat lengths
34     // torch::Tensor xysplat_lengths = torch::linalg_norm(xysplat, 2, 0, true, DATATYPE).to(DEVICE);
35     torch::Tensor xysplat_lengths = torch::linalg_norm(xysplat, 2, 0, true, torch::kFloat).to(DATATYPE);
36     if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 35 \n";
37
38     // finding azimuthal and elevation angles
39     torch::Tensor azimuthal_angles = torch::atan2(xysplat[1], xysplat[0]).to(DEVICE) * 180/PI;
40     azimuthal_angles = azimuthal_angles.reshape({1, azimuthal_angles.numel()});
41     torch::Tensor elevation_angles = torch::atan2(cartesian_vector[2], xysplat_lengths).to(DEVICE) * 180/PI;
42     // torch::Tensor rho_values = torch::linalg_norm(cartesian_vector, 2, 0, true, DATATYPE).to(DEVICE);
43     torch::Tensor rho_values = torch::linalg_norm(cartesian_vector, \
44                                                  2, 0, true, torch::kFloat).to(DATATYPE);
45     if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 42 \n";
46
47
48     // printing values for debugging
49     if (DEBUG_Cart2Sph){
50         std::cout<<"azimuthal_angles.shape = "; fPrintTensorSize(azimuthal_angles);
51         std::cout<<"elevation_angles.shape = "; fPrintTensorSize(elevation_angles);
52         std::cout<<"rho_values.shape = "; fPrintTensorSize(rho_values);
53     }
54     if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 51 \n";
55
56     // creating tensor to send back
57     torch::Tensor spherical_vector = torch::cat({azimuthal_angles, \
58                                                  elevation_angles, \
59                                                  rho_values}, 0).to(DEVICE);
60     if (DEBUG_Cart2Sph) std::cout<<"\t fCart2Sph: line 57 \n";
61
62     // returning the value
63     return spherical_vector;
64 }

```

8.3.2 Spherical Coordinates to Cartesian Coordinates

```

1 namespace svr {
2     // =====
3     template <typename T>
4     auto cart2sph(const std::vector<T> cartesian_vector){
5
6         // splatting the point onto xy-plane
7         auto xysplat {cartesian_vector};
8         xysplat[2] = 0;
9
10        // finding splat lengths
11        auto xysplat_lengths {norm(xysplat)};
12
13        // finding azimuthal and elevation angles
14        auto azimuthal_angles {svr::atan2(xysplat[1], xysplat[0]) * 180.00/std::numbers::pi};
15        auto elevation_angles {svr::atan2(cartesian_vector[2], xysplat_lengths) * 180.00/std::numbers::pi};
16        auto rho_values {norm(cartesian_vector)};
17
18        // creating tensor to send back
19        auto spherical_vector {std::vector<T>{azimuthal_angles,
20                                              elevation_angles,
21                                              rho_values}};
22
23        // moving it back
24        return std::move(spherical_vector);
25    }
26    // =====
27    template <typename T>
28    auto sph2cart(const std::vector<T> spherical_vector){
29
30        // creating cartesian vector
31        auto cartesian_vector {std::vector<T>(spherical_vector.size(), 0)};
32
33        // populating
34        cartesian_vector[0] = spherical_vector[2] * \
35                               cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
36                               cos(spherical_vector[0] * std::numbers::pi / 180.00);
37        cartesian_vector[1] = spherical_vector[2] * \
38                               cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
39                               sin(spherical_vector[0] * std::numbers::pi / 180.00);
40        cartesian_vector[2] = spherical_vector[2] * \
41                               sin(spherical_vector[1] * std::numbers::pi / 180.00);
42
43        // returning
44        return std::move(cartesian_vector);
45    }
46 }
47

```

8.3.3 Column-Wise Convolution

```

1  /* =====
2  Aim: Convolving the columns of two input matrices
3  =====*/
4  #include <ratio>
5  #include <stdexcept>
6  #include <torch/torch.h>
7
8  #pragma once
9
10 // hash-defines
11 #define PI 3.14159265
12 #define MYDEBUGFLAG false
13
14 #ifndef DEVICE
15     // #define DEVICE torch::kMPS
16     #define DEVICE torch::kCPU
17 #endif

```

```

18
19
20 void fConvolveColumns(torch::Tensor& inputMatrix, \
21                     torch::Tensor& kernelMatrix){
22
23
24     // printing shape
25     if(MYDEBUGFLAG) std::cout<<"inputMatrix.shape =
26         [<<inputMatrix.size(0)<<","<<inputMatrix.size(1)<<std::endl;
27     if(MYDEBUGFLAG) std::cout<<"kernelMatrix.shape =
28         [<<kernelMatrix.size(0)<<","<<kernelMatrix.size(1)<<std::endl;
29
30     // ensuring the two have the same number of columns
31     if (inputMatrix.size(1) != kernelMatrix.size(1)){
32         throw std::runtime_error("fConvolveColumns: arguments cannot have different number of columns");
33     }
34
35     // calculating length of final result
36     int final_length = inputMatrix.size(0) + kernelMatrix.size(0) - 1; if(MYDEBUGFLAG) std::cout<<"\t\t\t
37         fConvolveColumns: 27"<<std::endl;
38
39     // converting the two arguments to float since fft doesn't work with halves
40     inputMatrix = inputMatrix.to(torch::kFloat);
41     kernelMatrix = kernelMatrix.to(torch::kFloat);
42
43     // calculating FFT of the two matrices
44     torch::Tensor inputMatrix_FFT = torch::fft::fftn(inputMatrix, \
45         {final_length}, \
46         {0}); if(MYDEBUGFLAG) std::cout<<"\t\t\t fConvolveColumns:
47         32"<<std::endl;
48     torch::Tensor kernelMatrix_FFT = torch::fft::fftn(kernelMatrix, \
49         {final_length}, \
50         {0}); if(MYDEBUGFLAG) std::cout<<"\t\t\t fConvolveColumns:
51         35"<<std::endl;
52
53     // element-wise multiplying the two matrices
54     torch::Tensor MulProduct = torch::mul(inputMatrix_FFT, kernelMatrix_FFT); if(MYDEBUGFLAG)
55         std::cout<<"\t\t\t fConvolveColumns: 38"<<std::endl;
56
57     // finding the inverse FFT
58     torch::Tensor convolvedResult = torch::fft::ifftn(MulProduct, \
59         {MulProduct.size(0)}, \
60         {0}); if(MYDEBUGFLAG) std::cout<<"\t\t\t fConvolveColumns:
61         43"<<std::endl;
62
63     // bringing them back to the pipeline datatype
64     kernelMatrix = kernelMatrix.to(DATATYPE);
65
66     // over-riding the result with the input so that we can save memory
67     inputMatrix = convolvedResult.to(DATATYPE); if(MYDEBUGFLAG) std::cout<<"\t\t\t fConvolveColumns:
68         46"<<std::endl;
69
70 }

```

8.3.4 Buffer 2D

```

1  /* =====
2  Aim: Convolving the columns of two input matrices
3  =====*/
4  #include <stdexcept>
5  #include <torch/torch.h>
6
7  #pragma once
8
9  // hash-defines
10 #ifndef DEVICE
11     // #define DEVICE      torch::kMPS
12     #define DEVICE      torch::kCPU
13 #endif
14

```



```

15 // #define DEBUG_Buffer2D true
16 #define DEBUG_Buffer2D false
17
18
19 void fBuffer2D(torch::Tensor& inputMatrix,
20               int frame_size){
21
22     // ensuring the first dimension is 1.
23     if(inputMatrix.size(0) != 1){
24         throw std::runtime_error("fBuffer2D: The first-dimension must be 1 \n");
25     }
26
27     // padding with zeros in case it is not a perfect multiple
28     if(inputMatrix.size(1)%frame_size != 0){
29         // padding with zeros
30         int numberofzeroestoadd = frame_size - (inputMatrix.size(1) % frame_size);
31         if(DEBUG_Buffer2D) {
32             std::cout << "\t\t\t fBuffer2D: frame_size = " << frame_size <<
33                 std::endl;
34             std::cout << "\t\t\t fBuffer2D: inputMatrix.sizes().vec() = " << inputMatrix.sizes().vec() <<
35                 std::endl;
36             std::cout << "\t\t\t fBuffer2D: numberofzeroestoadd = " << numberofzeroestoadd << std::endl;
37         }
38
39         // creating zero matrix
40         torch::Tensor zeroMatrix = torch::zeros({inputMatrix.size(0), \
41             numberofzeroestoadd, \
42             inputMatrix.size(2)});
43         if(DEBUG_Buffer2D) std::cout<<"\t\t\t fBuffer2D: zeroMatrix.sizes() =
44             "<<zeroMatrix.sizes().vec()<<std::endl;
45
46         // adding the zero matrix
47         inputMatrix = torch::cat({inputMatrix, zeroMatrix}, 1);
48         if(DEBUG_Buffer2D) std::cout<<"\t\t\t fBuffer2D: inputMatrix.sizes().vec() =
49             "<<inputMatrix.sizes().vec()<<std::endl;
50     }
51
52     // calculating some parameters
53     // int num_frames = inputMatrix.size(1)/frame_size;
54     int num_frames = std::ceil(inputMatrix.size(1)/frame_size);
55     if(DEBUG_Buffer2D) std::cout << "\t\t\t fBuffer2D: inputMatrix.sizes = "<< inputMatrix.sizes().vec()<<
56         std::endl;
57     if(DEBUG_Buffer2D) std::cout << "\t\t\t fBuffer2D: framesize = " << frame_size << std::endl;
58     if(DEBUG_Buffer2D) std::cout << "\t\t\t fBuffer2D: num_frames = " << num_frames << std::endl;
59
60     // defining target shape and size
61     std::vector<int64_t> target_shape = {num_frames, \
62         frame_size, \
63         inputMatrix.size(2)};
64     std::vector<int64_t> target_strides = {frame_size * inputMatrix.size(2), \
65         inputMatrix.size(2), \
66         1};
67     if(DEBUG_Buffer2D) std::cout << "\t\t\t fBuffer2D: STATUS: created shape and strides"<< std::endl;
68
69     // creating the transformation
70     inputMatrix = inputMatrix.as_strided(target_shape, target_strides);
71 }

```

8.3.5 fAnglesToTensor

```

1 #include <torch/torch.h>
2 // function: angles to vector
3 torch::Tensor fAnglesToTensor(float azimuthal_angle,
4                               float elevation_angle)
5 {
6     // calculating tensor
7     torch::Tensor coordinateTensor = torch::tensor({cos(elevation_angle) * cos(azimuthal_angle),
8             cos(elevation_angle) * sin(azimuthal_angle),
9             sin(elevation_angle)}).view({3,1});

```

```
10
11 // returning value
12 return coordinateTensor;
13 }
```

8.3.6 fCalculateCosine

```
1 // including headerfiles
2 #include <torch/torch.h>
3
4 // function to calculate cosine of two tensors
5 torch::Tensor fCalculateCosine(torch::Tensor inputTensor1,
6                               torch::Tensor inputTensor2)
7 {
8     // column normalizing the the two signals
9     inputTensor1 = fColumnNormalize(inputTensor1);
10    inputTensor2 = fColumnNormalize(inputTensor2);
11
12    // finding their dot product
13    torch::Tensor dotProduct = inputTensor1 * inputTensor2;
14    torch::Tensor cosineBetweenVectors = torch::sum(dotProduct,
15                                                     0,
16                                                     true);
17
18    // returning the value
19    return cosineBetweenVectors;
20
21 }
```

8.4 Main Scripts

8.4.1 Signal Simulation

1.

```

1  /*=====
2  Aim: Signal Simulation
3  -----
4  =====*/
5
6  // including
7  #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/packages.h"
8  #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/config.h" // hash-defines
9  #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/classes.h" // class defs
10 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/setupscripts.h" // setup-scripts
11 #include "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/include/functions.h" // functions
12
13
14 // main-function
15 int main() {
16
17     // Ensuring no-gradients are built
18     NoGradGuard no_grad;
19
20     // Building Sea-floor
21     ScattererClass SeafloorScatter;
22     thread scatterThread_t(SeafloorSetup, \
23                           ref(SeafloorScatter));
24
25     // Building ULA
26     ULAClass ula_fls, ula_port, ula_starboard;
27     thread ulaThread_t(ULASetup, \
28                       ref(ula_fls), \
29                       ref(ula_port), \
30                       ref(ula_starboard));
31
32     // Building Transmitter
33     TransmitterClass transmitter_fls, transmitter_port, transmitter_starboard;
34     thread transmitterThread_t(TransmitterSetup,
35                               ref(transmitter_fls),
36                               ref(transmitter_port),
37                               ref(transmitter_starboard));
38
39     // recombining threads
40     scatterThread_t.join(); // making the scattetr population thread join back
41     ulaThread_t.join();    // making the ULA population thread join back
42     transmitterThread_t.join(); // making the transmitter population thread join back
43
44     // building AUV
45     AUVClass auv; // instantiating class object
46     AUVSetup(&auv); // populating
47
48     // attaching components to the AUV
49     auv.ULA_fls = ula_fls; // attaching ULA-FLS to AUV
50     auv.ULA_port = ula_port; // attaching ULA-Port to AUV
51     auv.ULA_starboard = ula_starboard; // attaching ULA-Starboard to AUV
52     auv.transmitter_fls = transmitter_fls; // attaching Transmitter-FLS to AUV
53     auv.transmitter_port = transmitter_port; // attaching Transmitter-Port to AUV
54     auv.transmitter_starboard = transmitter_starboard; // attaching Transmitter-Starboard to AUV
55
56     // storing
57     ScattererClass SeafloorScatter_deepcopy = SeafloorScatter;
58
59     // pre-computing the data-structures required for processing
60     auv.init();
61
62     // mimicking movement
63     int number_of_stophops = 4;
64     // if (true) return 0;
65     for(int i = 0; i<number_of_stophops; ++i){

```

```

66
67 // time measuring
68 auto start_time = high_resolution_clock::now();
69
70 // printing some spaces
71 PRINTSPACE; PRINTSPACE; PRINTLINE; cout<<"i = "<<i<<endl; PRINTLINE
72
73 // making the deep copy
74 ScattererClass SeafloorScatter = SeafloorScatter_deepcopy;
75
76 // signal simulation
77 auv.simulateSignal(SeafloorScatter);
78
79 // saving simulated signal
80 if (SAVETENSORS) {
81
82     // saving the signal matrix tensors
83     save(auv.ULA_fls.signalMatrix, \
84         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_fls.pt");
85     save(auv.ULA_port.signalMatrix, \
86         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_port.pt");
87     save(auv.ULA_starboard.signalMatrix, \
88         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/signalMatrix_starboard.pt");
89
90     // running python script
91     string script_to_run = \
92         "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/Python/Plot_SignalMatrix.py";
93     thread plotSignalMatrix_t(fRunSystemScriptInSeperateThread, \
94         script_to_run);
95     plotSignalMatrix_t.detach();
96
97 }
98
99
100 if (IMAGING_TOGGLE) {
101
102     // creating image from signals
103     auv.image();
104
105     // saving the tensors
106     if(SAVETENSORS){
107         // saving the beamformed images
108         save(auv.ULA_fls.beamformedImage, \
109             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/ULA_fls_image.pt");
110         // save(auv.ULA_port.beamformedImage, \
111             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/ULA_port_image.pt");
112         // save(auv.ULA_starboard.beamformedImage, \
113             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/ULA_starboard_image.pt");
114
115         // saving cartesian image
116         save(auv.ULA_fls.cartesianImage, \
117             "/Users/vrsreeganesh/Documents/GitHub/AUV/Code/C++/Assets/ULA_fls_cartesianImage.pt");
118
119         // // running python file
120         // system("python
121             /Users/vrsreeganesh/Documents/GitHub/AUV/Code/Python/Plot_BeamformedImage.py");
122         system("python /Users/vrsreeganesh/Documents/GitHub/AUV/Code/Python/Plot_cartesianImage.py");
123     }
124
125
126
127 // measuring and printing time taken
128 auto end_time = high_resolution_clock::now();
129 duration<double> time_duration = end_time - start_time;
130 PRINTDOTS; cout<<"Time taken (i = "<<i<<" = "<<time_duration.count()<<" seconds"<<endl; PRINTDOTS
131
132 // moving to next position
133 auv.step(0.5);
134
135 }
136
137
138

```

```
139
140
141
142
143     // returning
144     return 0;
145 }
```

Chapter 9

Reading

9.1 Primary Books

- 1.

9.2 Interesting Papers

Chapter 10

General Purpose Templated Functions

10.1 CSV File-Writes

```
1 // =====
2 template <typename T>
3 void fWriteVector(const vector<T>&      inputvector,
4                  const string&         filename){
5
6     // opening a file
7     std::ofstream fileobj(filename);
8     if (!fileobj) {return;}
9
10    // writing the real parts in the first column and the imaginary parts in the second column
11    if constexpr(std::is_same_v<T, std::complex<double>> ||
12                std::is_same_v<T, std::complex<float>> ||
13                std::is_same_v<T, std::complex<long double>>){
14        for(int i = 0; i<inputvector.size(); ++i){
15            // adding entry
16            fileobj << inputvector[i].real() << "+" << inputvector[i].imag() << "i";
17
18            // adding delimiter
19            if(i!=inputvector.size()-1) {fileobj << ",";}
20            else {fileobj << "\n";}
21        }
22    }
23    else{
24        for(int i = 0; i<inputvector.size(); ++i){
25            fileobj << inputvector[i];
26            if(i!=inputvector.size()-1) {fileobj << ",";}
27            else {fileobj << "\n";}
28        }
29    }
30
31    // return
32    return;
33 }
34 // Matrix writing =====
35 template <typename T>
36 auto fWriteMatrix(const std::vector<std::vector<T>> inputMatrix,
37                  const string                     filename){
38
39    // opening a file
40    std::ofstream fileobj(filename);
41
42    // writing
43    if (fileobj){
44        for(int i = 0; i<inputMatrix.size(); ++i){
45            for(int j = 0; j<inputMatrix[0].size(); ++j){
46                fileobj << inputMatrix[i][j];
47                if (j!=inputMatrix[0].size()-1) {fileobj << ",";}
48                else {fileobj << "\n";}
49            }
50        }
51    }
52 }
```

```

49     }
50 }
51 }
52 else{
53     cout << format("File-write to {} failed\n", filename);
54 }
55 }
56 }
57
58 template <>
59 auto fWriteMatrix(const std::vector<std::vector<std::complex<double>>> inputMatrix,
60                 const string filename){
61
62     // opening a file
63     std::ofstream fileobj(filename);
64
65     // writing
66     if (fileobj){
67         for(int i = 0; i<inputMatrix.size(); ++i){
68             for(int j = 0; j<inputMatrix[0].size(); ++j){
69                 fileobj << inputMatrix[i][j].real() << "+" << inputMatrix[i][j].imag() << "i";
70                 if (j!=inputMatrix[0].size()-1) {fileobj << ",";}
71                 else {fileobj << "\n";}
72             }
73         }
74     }
75     else{
76         cout << format("File-write to {} failed\n", filename);
77     }
78 }

```

10.2 abs

```

1 // =====
2 // y = abs(vector)
3 template <typename T>
4 auto abs(const std::vector<T>& input_vector)
5 {
6     // creating canvas
7     auto canvas {input_vector};
8
9     // calculating abs
10    std::transform(canvas.begin(),
11                  canvas.end(),
12                  canvas.begin(),
13                  [](auto& argx){return std::abs(argx);});
14
15    // returning
16    return std::move(canvas);
17 }
18 // =====
19 // y = abs(matrix)
20 template <typename T>
21 auto abs(const std::vector<std::vector<T>> input_matrix)
22 {
23     // creating canvas
24     auto canvas {input_matrix};
25
26     // applying element-wise abs
27    std::transform(input_matrix.begin(),
28                  input_matrix.end(),
29                  input_matrix.begin(),
30                  [](auto& argx){return std::abs(argx);});
31
32    // returning
33    return std::move(canvas);
34 }

```

10.3 Boolean Comparators

```

1 // =====
2 template <typename T, typename U>
3 auto operator<(const std::vector<T>& input_vector,
4               const U scalar)
5 {
6     // creating canvas
7     auto canvas {std::vector<bool>(input_vector.size())};
8
9     // transforming
10    std::transform(input_vector.begin(), input_vector.end(),
11                  canvas.begin(),
12                  [&scalar](const auto& argx){
13                      return argx < static_cast<T>(scalar);
14                  });
15
16    // returning
17    return std::move(canvas);
18 }
19 // =====
20 template <typename T, typename U>
21 auto operator<=(const std::vector<T>& input_vector,
22               const U scalar)
23 {
24     // creating canvas
25     auto canvas {std::vector<bool>(input_vector.size())};
26
27     // transforming
28    std::transform(input_vector.begin(), input_vector.end(),
29                  canvas.begin(),
30                  [&scalar](const auto& argx){
31                      return argx <= static_cast<T>(scalar);
32                  });
33
34    // returning
35    return std::move(canvas);
36 }
37 // =====
38 template <typename T, typename U>
39 auto operator>(const std::vector<T>& input_vector,
40               const U scalar)
41 {
42     // creating canvas
43     auto canvas {std::vector<bool>(input_vector.size())};
44
45     // transforming
46    std::transform(input_vector.begin(), input_vector.end(),
47                  canvas.begin(),
48                  [&scalar](const auto& argx){
49                      return argx > static_cast<T>(scalar);
50                  });
51
52    // returning
53    return std::move(canvas);
54 }
55 // =====
56 template <typename T, typename U>
57 auto operator>=(const std::vector<T>& input_vector,
58               const U scalar)
59 {
60     // creating canvas
61     auto canvas {std::vector<bool>(input_vector.size())};
62
63     // transforming
64    std::transform(input_vector.begin(), input_vector.end(),
65                  canvas.begin(),
66                  [&scalar](const auto& argx){
67                      return argx >= static_cast<T>(scalar);
68                  });
69
70    // returning
71    return std::move(canvas);
72 }

```

10.4 Concatenate Functions

```

1 // input = [vector, vector],
2 // output = [vector]
3 template <std::size_t axis, typename T>
4 auto concatenate(const std::vector<T>& input_vector_A,
5                 const std::vector<T>& input_vector_B) -> std::enable_if_t<axis == 1, std::vector<T> >
6 {
7     // creating canvas vector
8     auto num_elements {input_vector_A.size() + input_vector_B.size()};
9     auto canvas {std::vector<T>(num_elements, (T)0) };
10
11     // filling up the canvas
12     std::copy(input_vector_A.begin(), input_vector_A.end(),
13              canvas.begin());
14     std::copy(input_vector_B.begin(), input_vector_B.end(),
15              canvas.begin()+input_vector_A.size());
16
17     // moving it back
18     return std::move(canvas);
19 }
20
21 // =====
22 // input = [vector, vector],
23 // output = [matrix]
24 template <std::size_t axis, typename T>
25 auto concatenate(const std::vector<T>& input_vector_A,
26                 const std::vector<T>& input_vector_B) -> std::enable_if_t<axis == 0,
27                 std::vector<std::vector<T>> >
28 {
29     // throwing error dimensions
30     if (input_vector_A.size() != input_vector_B.size())
31         std::cerr << "concatenate:: incorrect dimensions \n";
32
33     // creating canvas
34     auto canvas {std::vector<std::vector<T>>(
35                 2, std::vector<T>(input_vector_A.size())
36             )};
37
38     // filling up the dimensions
39     std::copy(input_vector_A.begin(), input_vector_A.end(), canvas[0].begin());
40     std::copy(input_vector_B.begin(), input_vector_B.end(), canvas[1].begin());
41
42     // moving it back
43     return std::move(canvas);
44 }
45 // =====
46 // input = [vector, vector, vector],
47 // output = [matrix]
48 template <std::size_t axis, typename T>
49 auto concatenate(const std::vector<T>& input_vector_A,
50                 const std::vector<T>& input_vector_B,
51                 const std::vector<T>& input_vector_C) -> std::enable_if_t<axis == 0,
52                 std::vector<std::vector<T>> >
53 {
54     // throwing error dimensions
55     if (input_vector_A.size() != input_vector_B.size() ||
56         input_vector_A.size() != input_vector_C.size())
57         std::cerr << "concatenate:: incorrect dimensions \n";
58
59     // creating canvas
60     auto canvas {std::vector<std::vector<T>>(
61                 3, std::vector<T>(input_vector_A.size())
62             )};
63
64     // filling up the dimensions
65     std::copy(input_vector_A.begin(), input_vector_A.end(), canvas[0].begin());
66     std::copy(input_vector_B.begin(), input_vector_B.end(), canvas[1].begin());
67     std::copy(input_vector_C.begin(), input_vector_C.end(), canvas[2].begin());
68
69     // moving it back
70     return std::move(canvas);
71 }

```

```

71 }
72 // =====
73 // input = [matrix, vector],
74 // output = [matrix]
75 template <std::size_t axis, typename T>
76 auto concatenate(const std::vector<std::vector<T>>& input_matrix,
77                 const std::vector<T> input_vector) -> std::enable_if_t<axis == 0,
78                 std::vector<std::vector<T>> >
79 {
80     // creating canvas
81     auto canvas {input_matrix};
82
83     // adding to the canvas
84     canvas.push_back(input_vector);
85
86     // returning
87     return std::move(canvas);
88 }

```

10.5 Conjugate

```

1 namespace svr {
2     // =====
3     template <typename T>
4     auto conj(const std::vector<T>& input_vector)
5     {
6         // creating canvas
7         auto canvas {std::vector<T>(input_vector.size())};
8
9         // calculating conjugates
10        std::for_each(canvas.begin(), canvas.end(),
11                      [](auto& argx){argx = std::conj(argx);});
12
13        // returning
14        return std::move(canvas);
15    }
16 }

```

10.6 Convolution

```

1 namespace svr {
2     // =====
3     template <typename T1, typename T2>
4     auto conv1D(const std::vector<T1>& input_vector_A,
5                const std::vector<T2>& input_vector_B)
6     {
7         // resulting type
8         using T3 = decltype(std::declval<T1>() * std::declval<T2>());
9
10        // creating canvas
11        auto canvas_length {input_vector_A.size() + input_vector_B.size() - 1};
12
13        // calculating fft of two arrays
14        auto fft_A {svr::fft(input_vector_A, canvas_length)};
15        auto fft_B {svr::fft(input_vector_B, canvas_length)};
16
17        // element-wise multiplying the two matrices
18        auto fft_AB {fft_A * fft_B};
19
20        // finding inverse FFT
21        auto convolved_result {ifft(fft_AB)};
22
23        // returning
24        return std::move(convolved_result);
25    }
26 }

```

27 }

10.7 Coordinate Change

```

1 namespace svr {
2     // =====
3     template <typename T>
4     auto cart2sph(const std::vector<T> cartesian_vector){
5
6         // splatting the point onto xy-plane
7         auto xysplat {cartesian_vector};
8         xysplat[2] = 0;
9
10        // finding splat lengths
11        auto xysplat_lengths {norm(xysplat)};
12
13        // finding azimuthal and elevation angles
14        auto azimuthal_angles {svr::atan2(xysplat[1], xysplat[0]) * 180.00/std::numbers::pi};
15        auto elevation_angles {svr::atan2(cartesian_vector[2], xysplat_lengths) * 180.00/std::numbers::pi};
16        auto rho_values {norm(cartesian_vector)};
17
18        // creating tensor to send back
19        auto spherical_vector {std::vector<T>{azimuthal_angles,
20                                                elevation_angles,
21                                                rho_values}};
22
23        // moving it back
24        return std::move(spherical_vector);
25    }
26    // =====
27    template <typename T>
28    auto sph2cart(const std::vector<T> spherical_vector){
29
30        // creating cartesian vector
31        auto cartesian_vector {std::vector<T>(spherical_vector.size(), 0)};
32
33        // populating
34        cartesian_vector[0] = spherical_vector[2] * \
35                               cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
36                               cos(spherical_vector[0] * std::numbers::pi / 180.00);
37        cartesian_vector[1] = spherical_vector[2] * \
38                               cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
39                               sin(spherical_vector[0] * std::numbers::pi / 180.00);
40        cartesian_vector[2] = spherical_vector[2] * \
41                               sin(spherical_vector[1] * std::numbers::pi / 180.00);
42
43        // returning
44        return std::move(cartesian_vector);
45    }
46 }
47 }

```

10.8 Cosine

```

1 // =====
2 // y = cos(input_vector)
3 template <typename T>
4 auto cos(const std::vector<T>& input_vector)
5 {
6     // created canvas
7     auto canvas {input_vector};
8
9     // calling the function
10    std::transform(input_vector.begin(), input_vector.end(),
11                  canvas.begin(),
12                  [](auto& argx){return std::cos(argx);});

```

```

13
14     // returning the output
15     return std::move(canvas);
16 }
17 // =====
18 // y = cosd(input_vector)
19 template <typename T>
20 auto cosd(std::vector<T> input_vector)
21 {
22     // created canvas
23     auto canvas {input_vector};
24
25     // calling the function
26     std::transform(input_vector.begin(),
27                   input_vector.end(),
28                   input_vector.begin(),
29                   [](const auto& argx){return std::cos(argx * 180.00/std::numbers::pi);});
30
31     // returning the output
32     return std::move(canvas);
33 }

```

10.9 Data Structures

```

1 struct TreeNode {
2     int val;
3     TreeNode *left;
4     TreeNode *right;
5     TreeNode() : val(0), left(nullptr), right(nullptr) {}
6     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
7     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
8 };
9
10
11 struct ListNode {
12     int val;
13     ListNode *next;
14     ListNode() : val(0), next(nullptr) {}
15     ListNode(int x) : val(x), next(nullptr) {}
16     ListNode(int x, ListNode *next) : val(x), next(next) {}
17 };

```

10.10 Editing Index Values

```

1 // =====
2 template <typename T, typename BooleanVector, typename U>
3 auto edit(std::vector<T>& input_vector,
4          BooleanVector bool_vector,
5          const U scalar)
6 {
7     // throwing an error
8     if (input_vector.size() != bool_vector.size())
9         std::cerr << "edit: incompatible size\n";
10
11     // overwriting input-vector
12     std::transform(input_vector.begin(), input_vector.end(),
13                   bool_vector.begin(),
14                   input_vector.begin(),
15                   [&scalar](auto& argx, auto argy){
16                       if(argy == true) {return static_cast<T>(scalar);}
17                       else {return argx;}
18                   });
19
20     // no-returns since in-place
21 }

```

10.11 Equality

```

1 // =====
2 template <typename T, typename U>
3 auto operator==(const std::vector<T>& input_vector,
4                 const U& scalar)
5 {
6     // setting up canvas
7     auto canvas {std::vector<bool>(input_vector.size())};
8
9     // writing to canvas
10    std::transform(input_vector.begin(), input_vector.end(),
11                  canvas.begin(),
12                  [&scalar](const auto& argx){
13                      return argx == scalar;
14                  });
15
16    // returning
17    return std::move(canvas);
18 }

```

10.12 Exponentiate

```

1 // y = abs(vector)
2 template <typename T>
3 auto exp(const std::vector<T>& input_vector)
4 {
5     // creating canvas
6     auto canvas {input_vector};
7
8     // transforming
9     std::transform(canvas.begin(), canvas.end(),
10                   canvas.begin(),
11                   [](auto& argx){return std::exp(argx);});
12
13    // returning
14    return std::move(canvas);
15 }

```

10.13 FFT

```

1 namespace svr {
2     // =====
3     // For type-deductions
4     template <typename T>
5     struct fft_result_type;
6
7     // specializations
8     template <> struct fft_result_type<double>{
9         using type = std::complex<double>;
10    };
11    template <> struct fft_result_type<std::complex<double>>{
12        using type = std::complex<double>;
13    };
14    template <> struct fft_result_type<float>{
15        using type = std::complex<float>;
16    };
17    template <> struct fft_result_type<std::complex<float>>{
18        using type = std::complex<float>;
19    };
20
21    template <typename T>
22    using fft_result_t = typename fft_result_type<T>::type;
23
24    // // =====

```

```

25 // // y = fft(x, nfft)
26 // template<typename T>
27 // auto fft(const std::vector<T>& input_vector,
28 //          const size_t nfft)
29 // {
30
31 //     svr::Timer timer("fft");
32
33 //     // throwing an error
34 //     if (nfft < input_vector.size()) {std::cerr << "size-mismatch\n";}
35 //     if (nfft <= 0) {std::cerr << "size-mismatch\n";}
36
37 //     // fetching data-type
38 //     using RType = fft_result_t<T>;
39
40 //     // canvas instantiation
41 //     std::vector<RType> canvas(nfft);
42
43 //     // building time-only basis
44 //     std::vector<RType>
45 //     basiswithoutfrequency {linspace(static_cast<RType>(0),
46 //                                     static_cast<RType>(nfft-1),
47 //                                     nfft)};
48 //     auto lambda_basiswithoutfrequency = [&basiswithoutfrequency](RType& arg){
49 //         return std::exp(-1.00 * 1i * 2.00 *
50 //                         std::numbers::pi * static_cast<RType>(arg) / \
51 //                         static_cast<RType>(basiswithoutfrequency.size()));
52 //     };
53 //     std::transform(basiswithoutfrequency.begin(), basiswithoutfrequency.end(),
54 //                   basiswithoutfrequency.begin(),
55 //                   lambda_basiswithoutfrequency);
56
57 //     // building basis vectors
58 //     auto bases_vectors {std::vector<std::vector<RType>>()};
59 //     for(auto i = 0; i < nfft; ++i){
60 //         // making a copy of the bases-without-frequency
61 //         auto temp {basiswithoutfrequency};
62 //         // exponentiating basis with frequency
63 //         std::transform(temp.begin(), temp.end(),
64 //                        temp.begin(),
65 //                        [&i](auto& argx){return std::pow(argx, i);});
66 //         // pushing to end of bases-vectors
67 //         bases_vectors.push_back(std::move(temp));
68 //     }
69
70 //     // projecting input-array onto fourier bases
71 //     auto finaloutput {std::vector<RType>(nfft, 0)};
72 //     auto nfft_sqrt {static_cast<RType>(std::sqrt(nfft))};
73 //     #pragma omp parallel for
74 //     for(int i = 0; i < nfft; ++i){
75 //         // projecting input-vector with
76 //         finaloutput[i] = std::inner_product(input_vector.begin(), input_vector.end(),
77 //                                             bases_vectors[i].begin(),
78 //                                             RType(0),
79 //                                             std::plus<RType>(),
80 //                                             [&nfft_sqrt](const auto& argx,
81 //                                                         const auto& argy){
82 //                                                 return static_cast<RType>(argx) *
83 //                                                 static_cast<RType>(argy) / nfft_sqrt;
84 //                                             });
85 //     }
86 //     // returning finaloutput
87 //     return std::move(finaloutput);
88 // }
89
90 // =====
91 // y = fft(x, nfft)
92 template<typename T>
93 auto fft(const std::vector<T>& input_vector,
94          const size_t nfft)
95 {
96 //     // throwing an error
97 //     if (nfft < input_vector.size()) {std::cerr << "size-mismatch\n";}
98 //     if (nfft <= 0) {std::cerr << "size-mismatch\n";}

```

```

99     // fetching data-type
100     using RType = fft_result_t<T>;
101     using baseType = std::conditional_t<std::is_same_v<T, std::complex<double>>,
102                                     double,
103                                     T>;
104
105     // canvas instantiation
106     std::vector<RType> canvas(nfft);
107     auto nfft_sqrt = {static_cast<RType>(std::sqrt(nfft))};
108     auto finaloutput = {std::vector<RType>(nfft, 0)};
109
110     // calculating index by index
111     for(int frequency_index = 0; frequency_index<nfft; ++frequency_index){
112         RType accumulate_value;
113         for(int signal_index = 0; signal_index < input_vector.size(); ++signal_index){
114             accumulate_value += \
115                 static_cast<RType>(input_vector[signal_index]) * \
116                 static_cast<RType>(std::exp(-1.00 * std::numbers::pi * \
117                                         (static_cast<baseType>(frequency_index)/static_cast<baseType>(nfft))
118                                         * \
119                                         static_cast<baseType>(signal_index)));
120             finaloutput[frequency_index] = accumulate_value / nfft_sqrt;
121         }
122     }
123
124     // returning
125     return std::move(finaloutput);
126 }
127
128
129
130
131
132
133
134
135
136
137
138 // // =====
139 // // y = ifft(x)
140 // template<typename T>
141 // auto ifft(const std::vector<T>& input_vector)
142 // {
143 //     svr::Timer timer00("ifft");
144 //
145 //     // fetching nfft
146 //     auto nfft = {input_vector.size()};
147 //
148 //     // fetching data-type
149 //     using RType = fft_result_t<T>;
150 //
151 //     // canvas instantiation
152 //     std::vector<RType> canvas(nfft);
153 //
154 //     // building time-only basis
155 //     std::vector<RType>
156 //     basiswithoutfrequency {linspace(static_cast<RType>(0),
157 //                                     static_cast<RType>(nfft-1),
158 //                                     nfft)};
159 //     auto lambda_basiswithoutfrequency = [&basiswithoutfrequency](RType& arg){
160 //         return std::exp(1.00 * 1i * 2.00 * \
161 //                         std::numbers::pi * static_cast<RType>(arg) / \
162 //                         static_cast<RType>(basiswithoutfrequency.size()));
163 //     };
164 //     std::transform(basiswithoutfrequency.begin(), basiswithoutfrequency.end(),
165 //                   basiswithoutfrequency.begin(),
166 //                   lambda_basiswithoutfrequency);
167 //
168 //     // building basis vectors
169 //     auto bases_vectors {std::vector<std::vector<RType>>()};
170 //     for(auto i = 0; i < nfft; ++i){
171 //         // making a copy of the bases-without-frequency
172 //         auto temp {basiswithoutfrequency};

```



```

173 //      // exponentiating basis with frequency
174 //      std::transform(temp.begin(), temp.end(),
175 //                      temp.begin(),
176 //                      [&i](auto& argx){return std::pow(argx, i);});
177 //      // pushing to end of bases-vectors
178 //      bases_vectors.push_back(std::move(temp));
179 //  }
180
181 //  // projecting input-array onto fourier bases
182 //  auto finaloutput {std::vector<RType>(nfft, 0)};
183 //  auto nfft_sqrt {static_cast<RType>(std::sqrt(nfft))};
184 //  #pragma omp parallel for
185 //  for(int i = 0; i < nfft; ++i){
186 //      // projecting input-vector with
187 //      finaloutput[i] = std::inner_product(input_vector.begin(), input_vector.end(),
188 //                                          bases_vectors[i].begin(),
189 //                                          RType(0),
190 //                                          std::plus<RType>(),
191 //                                          [&nfft_sqrt](const auto& argx,
192 //                                                    const auto& argy){
193 //                                              return static_cast<RType>(argx) *
194 //                                              static_cast<RType>(argy) / nfft_sqrt;
195 //                                          });
196 //  }
197 //  // returning finaloutput
198 //  return std::move(finaloutput);
199 // }
200
201 // =====
202 // y = ifft(x, nfft)
203 template<typename T>
204 auto ifft(const std::vector<T>& input_vector)
205 {
206     // fetching data-type
207     using RType = fft_result_t<T>;
208     using baseType = std::conditional_t<std::is_same_v<T, std::complex<double>>,
209                                         double,
210                                         T>;
211
212     // setup
213     auto nfft {input_vector.size()};
214
215     // canvas instantiation
216     std::vector<RType> canvas(nfft);
217     auto nfft_sqrt {static_cast<RType>(std::sqrt(nfft))};
218     auto finaloutput {std::vector<RType>(nfft, 0)};
219
220     // calculating index by index
221     for(int frequency_index = 0; frequency_index < nfft; ++frequency_index){
222         RType accumulate_value;
223         for(int signal_index = 0; signal_index < input_vector.size(); ++signal_index){
224             accumulate_value += \
225                 static_cast<RType>(input_vector[signal_index]) * \
226                 static_cast<RType>(std::exp(1.00 * std::numbers::pi * \
227                                             (static_cast<baseType>(frequency_index)/static_cast<baseType>(nfft))
228                                             * \
229                                             static_cast<baseType>(signal_index)));
230         }
231         finaloutput[frequency_index] = accumulate_value / nfft_sqrt;
232     }
233
234     // returning
235     return std::move(finaloutput);
236 }

```

10.14 Flipping Containers

```

1 namespace svr {
2 // =====

```

```

3  template <typename T>
4  auto fliplr(const std::vector<T>& input_vector)
5  {
6      // creating canvas
7      auto canvas {input_vector};
8
9      // rewriting
10     std::reverse(canvas.begin(), canvas.end());
11
12     // returning
13     return std::move(canvas);
14 }
15 }

```

10.15 Indexing

```

1  namespace svr {
2      // =====
3      template <typename T1, typename T2>
4      auto index(const std::vector<T1>& input_vector,
5                  const std::vector<T2>& indices_to_sample)
6      {
7          // creating canvas
8          auto canvas {std::vector<T1>(indices_to_sample.size(), 0)};
9
10         // copying the associated values
11         for(int i = 0; i < indices_to_sample.size(); ++i){
12             auto source_index {indices_to_sample[i]};
13             if(source_index < input_vector.size()){
14                 canvas[i] = input_vector[source_index];
15             }
16             else
17                 cout << "svr::index | source_index !< input_vector.size()\n";
18         }
19
20         // returning
21         return std::move(canvas);
22     }
23 }

```

10.16 Linspace

```

1  // in-place
2  template <typename T>
3  auto linspace(auto& input,
4                auto startvalue,
5                auto endvalue,
6                auto numpoints) -> void
7  {
8      auto stepsize = static_cast<T>(endvalue - startvalue)/static_cast<T>(numpoints-1);
9      for(int i = 0; i<input.size(); ++i) {input[i] = startvalue + i*stepsize;}
10 };
11 // in-place
12 template <typename T>
13 auto linspace(vector<complex<T>>& input,
14               auto startvalue,
15               auto endvalue,
16               auto numpoints) -> void
17 {
18     auto stepsize = static_cast<T>(endvalue - startvalue)/static_cast<T>(numpoints-1);
19     for(int i = 0; i<input.size(); ++i) {
20         input[i] = startvalue + static_cast<T>(i)*stepsize;
21     }
22 };
23
24 // return-type

```

```

25 template <typename T>
26 auto linspace(T          startvalue,
27              T          endvalue,
28              size_t      numpoints)
29 {
30     vector<T> input(numpoints);
31     auto stepsize = static_cast<T>(endvalue - startvalue)/static_cast<T>(numpoints-1);
32
33     for(int i = 0; i<input.size(); ++i) {input[i] = startvalue + static_cast<T>(i)*stepsize;}
34
35     return input;
36 };
37
38 // return-type
39 template <typename T, typename U>
40 auto linspace(T          startvalue,
41              U          endvalue,
42              size_t      numpoints)
43 {
44     vector<double> input(numpoints);
45     auto stepsize = static_cast<double>(endvalue - startvalue)/static_cast<double>(numpoints-1);
46
47     for(int i = 0; i<input.size(); ++i) {input[i] = startvalue + i*stepsize;}
48
49     return input;
50 };

```

10.17 Max

```

1  template <std::size_t axis, typename T>
2  auto max(const std::vector<std::vector<T>> input_matrix) -> std::enable_if_t<axis == 1,
   std::vector<std::vector<T>> >
3  {
4      // setting up canvas
5      auto canvas {std::vector<std::vector<T>>(input_matrix.size(),std::vector<T>(1))};
6
7      // filling up the canvas
8      for(auto row = 0; row < input_matrix.size(); ++row)
9          canvas[row][0] = *(std::max_element(input_matrix[row].begin(), input_matrix[row].end()));
10
11     // returning
12     return std::move(canvas);
13 }

```

10.18 Meshgrid

```

1  // =====
2  template <typename T>
3  auto meshgrid(const std::vector<T>& x,
4               const std::vector<T>& y)
5  {
6
7      // creating and filling x-grid
8      std::vector<std::vector<T>> xcanvas(y.size(), std::vector<T>(x.size(), 0));
9      for(auto row = 0; row < y.size(); ++row)
10         std::copy(x.begin(), x.end(), xcanvas[row].begin());
11
12     // creating and filling y-grid
13     std::vector<std::vector<T>> ycanvas(y.size(), std::vector<T>(x.size(), 0));
14     for(auto col = 0; col < x.size(); ++col)
15         for(auto row = 0; row < y.size(); ++row)
16             ycanvas[row][col] = y[row];
17
18     // returning
19     return std::move(std::pair{xcanvas, ycanvas});
20 }

```

```

21 }
22 // =====
23 template <typename T>
24 auto meshgrid(std::vector<T>&& x,
25              std::vector<T>&& y)
26 {
27
28     // creating and filling x-grid
29     std::vector<std::vector<T>> xcanvas(y.size(), std::vector<T>(x.size(), 0));
30     for(auto row = 0; row < y.size(); ++row)
31         std::copy(x.begin(), x.end(), xcanvas[row].begin());
32
33     // creating and filling y-grid
34     std::vector<std::vector<T>> ycanvas(y.size(), std::vector<T>(x.size(), 0));
35     for(auto col = 0; col < x.size(); ++col)
36         for(auto row = 0; row < y.size(); ++row)
37             ycanvas[row][col] = y[row];
38
39     // returning
40     return std::move(std::pair{xcanvas, ycanvas});
41 }
42 }

```

10.19 Minimum

```

1 template <std::size_t axis, typename T>
2 auto min(std::vector<std::vector<T>> input_matrix) -> std::enable_if_t<axis == 1,
3             std::vector<std::vector<T>>> >
4 {
5     // creating canvas
6     auto canvas {std::vector<std::vector<T>>(input_matrix.size(), std::vector<T>(1))};
7
8     // storing the values
9     for(auto row = 0; row < input_matrix.size(); ++row)
10         canvas[row][0] = *(std::min_element(input_matrix[row].begin(), input_matrix[row].end()));
11
12     // returning the value
13     return std::move(canvas);
14 }

```

10.20 Norm

```

1 // =====
2 template <typename T>
3 auto norm(const std::vector<T>& input_vector)
4 {
5     return std::sqrt(std::inner_product(input_vector.begin(), input_vector.end(),
6                                         input_vector.begin(),
7                                         (T)0));
8 }
9
10
11
12 /*
13 Templates to create
14 - matrix and norm-axis
15 - axis instantiated std::vector<T>
16 */

```

10.21 Division

```

1 // =====
2 // matrix division with scalars
3 template <typename T>
4 auto operator/(const std::vector<T>& input_vector,
5               const T& input_scalar)
6 {
7     // creating canvas
8     auto canvas {input_vector};
9
10    // filling canvas
11    std::transform(canvas.begin(), canvas.end(),
12                  canvas.begin(),
13                  [&input_scalar](const auto& argx){
14                      return static_cast<double>(argx) / static_cast<double>(input_scalar);
15                  });
16
17    // returning value
18    return std::move(canvas);
19 }
20 // =====
21 // matrix division with scalars
22 template <typename T>
23 auto operator/=(const std::vector<T>& input_vector,
24                const T& input_scalar)
25 {
26     // creating canvas
27     auto canvas {input_vector};
28
29     // filling canvas
30     std::transform(canvas.begin(), canvas.end(),
31                   canvas.begin(),
32                   [&input_scalar](const auto& argx){
33                       return static_cast<double>(argx) / static_cast<double>(input_scalar);
34                   });
35
36     // returning value
37     return std::move(canvas);
38 }

```

10.22 Addition

```

1 // =====
2 // y = vector + vector
3 template <typename T>
4 std::vector<T> operator+(const std::vector<T>& a,
5                        const std::vector<T>& b)
6 {
7     // Identify which is bigger
8     const auto& big = (a.size() > b.size()) ? a : b;
9     const auto& small = (a.size() > b.size()) ? b : a;
10
11    std::vector<T> result = big; // copy the bigger one
12
13    // Add elements from the smaller one
14    for (size_t i = 0; i < small.size(); ++i) {
15        result[i] += small[i];
16    }
17
18    return result;
19 }
20 // =====
21 // y = vector + vector
22 template <typename T>
23 std::vector<T>& operator+=(std::vector<T>& a,
24                          const std::vector<T>& b) {
25
26    const auto& small = (a.size() < b.size()) ? a : b;
27    const auto& big = (a.size() < b.size()) ? b : a;
28
29    // If b is bigger, resize 'a' to match

```

```

30     if (a.size() < b.size())                {a.resize(b.size());}
31
32     // Add elements
33     for (size_t i = 0; i < small.size(); ++i) {a[i] += b[i];}
34
35     // returning elements
36     return a;
37 }
38 // =====
39 // y = matrix + matrix
40 template <typename T>
41 std::vector<std::vector<T>> operator+(const std::vector<std::vector<T>>& a,
42                                     const std::vector<std::vector<T>>& b)
43 {
44     // fetching dimensions
45     const auto& num_rows_A    {a.size()};
46     const auto& num_cols_A    {a[0].size()};
47     const auto& num_rows_B    {b.size()};
48     const auto& num_cols_B    {b[0].size()};
49
50     // choosing the three different metrics
51     if (num_rows_A != num_rows_B && num_cols_A != num_cols_B){
52         cout << format("a.dimensions = [{},{}], b.shape = [{},{}]\n",
53                        num_rows_A, num_cols_A,
54                        num_rows_B, num_cols_B);
55         std::cerr << "dimensions don't match\n";
56     }
57
58     // creating canvas
59     auto canvas {std::vector<std::vector<T>>(
60         std::max(num_rows_A, num_rows_B),
61         std::vector<T>(std::max(num_cols_A, num_cols_B), (T)0.00)
62     )};
63
64     // performing addition
65     if (num_rows_A == num_rows_B && num_cols_A == num_cols_B){
66         for(auto row = 0; row < num_rows_A; ++row){
67             std::transform(a[row].begin(), a[row].end(),
68                            b[row].begin(),
69                            canvas[row].begin(),
70                            std::plus<T>());
71         }
72     }
73     else if(num_rows_A == num_rows_B){
74
75         // if number of columns are different, check if one of the cols are one
76         const auto min_num_cols {std::min(num_cols_A, num_cols_B)};
77         if (min_num_cols != 1) {std::cerr<< "Operator+: unable to broadcast\n";}
78         const auto max_num_cols {std::max(num_cols_A, num_cols_B)};
79
80         // using references to tag em differently
81         const auto& big_matrix    {num_cols_A > num_cols_B ? a : b};
82         const auto& small_matrix  {num_cols_A < num_cols_B ? a : b};
83
84         // Adding to canvas
85         for(auto row = 0; row < canvas.size(); ++row){
86             std::transform(big_matrix[row].begin(), big_matrix[row].end(),
87                            canvas[row].begin(),
88                            [&small_matrix,
89                             &row](const auto& argx){
90                                 return argx + small_matrix[row][0];
91                             });
92         }
93     }
94     else if(num_cols_A == num_cols_B){
95
96         // check if the smallest column-number is one
97         const auto min_num_rows {std::min(num_rows_A, num_rows_B)};
98         if(min_num_rows != 1) {std::cerr << "Operator+ : unable to broadcast\n";}
99         const auto max_num_rows {std::max(num_rows_A, num_rows_B)};
100
101         // using references to differentiate the two matrices
102         const auto& big_matrix    {num_rows_A > num_rows_B ? a : b};
103         const auto& small_matrix  {num_rows_A < num_rows_B ? a : b};

```

```

105     // adding to canvas
106     for(auto row = 0; row < canvas.size(); ++row){
107         std::transform(big_matrix[row].begin(), big_matrix[row].end(),
108             small_matrix[0].begin(),
109             canvas[row].begin(),
110             [](const auto& argx, const auto& argy){
111                 return argx + argy;
112             });
113     }
114 }
115 else {
116     PRINTLINE PRINTLINE PRINTLINE PRINTLINE PRINTLINE
117     cout << format("check this again \n");
118 }
119
120 // returning
121 return std::move(canvas);
122 }
123 // =====
124 // y = vector + scalar
125 template <typename T>
126 auto operator+(const std::vector<T>& input_vector,
127               const T scalar)
128 {
129     // creating canvas
130     auto canvas {input_vector};
131
132     // adding scalar to the canvas
133     std::transform(canvas.begin(), canvas.end(),
134         canvas.begin(),
135         [&scalar](auto& argx){return argx + scalar;});
136
137     // returning canvas
138     return std::move(canvas);
139 }
140 // =====
141 // y = scalar + vector
142 template <typename T>
143 auto operator+(const T scalar,
144               const std::vector<T>& input_vector)
145 {
146     // creating canvas
147     auto canvas {input_vector};
148
149     // adding scalar to the canvas
150     std::transform(canvas.begin(), canvas.end(),
151         canvas.begin(),
152         [&scalar](auto& argx){return argx + scalar;});
153
154     // returning canvas
155     return std::move(canvas);
156 }

```

10.23 Multiplication (Element-wise)

```

1 // scalar * vector =====
2 template <typename T>
3 auto operator*(const T scalar,
4               const std::vector<T>& input_vector)
5 {
6     // creating canvas
7     auto canvas {input_vector};
8     // performing operation
9     std::for_each(canvas.begin(), canvas.end(),
10         [&scalar](auto& argx){argx = argx * scalar;});
11     // returning
12     return std::move(canvas);
13 }
14
15 // scalar * vector =====

```

```

16 // template <typename T1, typename T2>
17 template <typename T1, typename T2,
18         typename = std::enable_if_t<!std::is_same_v<std::decay_t<T1>, std::vector<T2>>>>
19 auto operator*(const T1 scalar,
20               const vector<T2>& input_vector)
21 {
22     // fetching final-type
23     using T3 = decltype(std::declval<T1>() * std::declval<T2>());
24     // creating canvas
25     auto canvas {std::vector<T3>(input_vector.size())};
26     // multiplying
27     std::transform(input_vector.begin(), input_vector.end(),
28                   canvas.begin(),
29                   [&scalar](auto& argx){
30                       return static_cast<T3>(scalar) * static_cast<T3>(argx);
31                   });
32     // returning
33     return std::move(canvas);
34 }
35
36 // vector * scalar =====
37 template <typename T>
38 auto operator*(const std::vector<T>& input_vector,
39               const T scalar)
40 {
41     // creating canvas
42     auto canvas {input_vector};
43     // multiplying
44     std::for_each(canvas.begin(), canvas.end(),
45                   [&scalar](auto& argx){
46                       argx = argx * scalar;
47                   });
48     // returning
49     return std::move(canvas);
50 }
51
52 // vector * vector =====
53 template <typename T>
54 auto operator*(const std::vector<T>& input_vector_A,
55               const std::vector<T>& input_vector_B)
56 {
57     // throwing error: size-disparity
58     if (input_vector_A.size() != input_vector_B.size()) {std::cerr << "operator*: size disparity \n";}
59
60     // creating canvas
61     auto canvas {input_vector_A};
62
63     // element-wise multiplying
64     std::transform(input_vector_B.begin(), input_vector_B.end(),
65                   canvas.begin(),
66                   canvas.begin(),
67                   [](const auto& argx, const auto& argy){
68                       return argx * argy;
69                   });
70
71     // moving it back
72     return std::move(canvas);
73 }
74 template <typename T1, typename T2>
75 auto operator*(const std::vector<T1>& input_vector_A,
76               const std::vector<T2>& input_vector_B)
77 {
78
79     // checking size disparity
80     if (input_vector_A.size() != input_vector_B.size())
81         std::cerr << "operator*: error, size-disparity \n";
82
83     // figuring out resulting data type
84     using T3 = decltype(std::declval<T1>() * std::declval<T2>());
85
86     // creating canvas
87     auto canvas {std::vector<T3>(input_vector_A.size())};
88
89     // performing multiplications
90     std::transform(input_vector_A.begin(), input_vector_A.end(),

```



```

91         input_vector_B.begin(),
92         canvas.begin(),
93         [](const auto& argx,
94            const auto& argy){
95             return static_cast<T3>(argx) * static_cast<T3>(argy);
96         });
97
98     // returning
99     return std::move(canvas);
100 }
101
102 // scalar * matrix =====
103 template <typename T>
104 auto operator*(T scalar,
105               const std::vector<std::vector<T>>& inputMatrix)
106 {
107     std::vector<std::vector<T>> temp {inputMatrix};
108     for(int i = 0; i<inputMatrix.size(); ++i){
109         std::transform(inputMatrix[i].begin(),
110                        inputMatrix[i].end(),
111                        temp[i].begin(),
112                        [&scalar](T x){return scalar * x;});
113     }
114     return temp;
115 }
116
117 // matrix * matrix =====
118 template <typename T>
119 auto operator*(const std::vector<std::vector<T>>& A,
120               const std::vector<std::vector<T>>& B) -> std::vector<std::vector<T>>
121 {
122     // Case 1: element-wise multiplication
123     if (A.size() == B.size() && A[0].size() == B[0].size()) {
124         std::vector<std::vector<T>> C(A.size(), std::vector<T>(A[0].size()));
125         for (std::size_t row = 0; row < A.size(); ++row) {
126             std::transform(A[row].begin(), A[row].end(),
127                            B[row].begin(),
128                            C[row].begin(),
129                            [](const auto& x, const auto& y){ return x * y; });
130         }
131         return C;
132     }
133
134     // Case 2: broadcast column vector
135     else if (A.size() == B.size() && B[0].size() == 1) {
136         std::vector<std::vector<T>> C(A.size(), std::vector<T>(A[0].size()));
137         for (std::size_t row = 0; row < A.size(); ++row) {
138             std::transform(A[row].begin(), A[row].end(),
139                            C[row].begin(),
140                            [&](const auto& x){ return x * B[row][0]; });
141         }
142         return C;
143     }
144
145     // case 3: when second matrix contains just one row
146     // case 4: when first matrix is just one column
147     // case 5: when second matrix is just one column
148
149     // Otherwise, invalid
150     else {
151         throw std::runtime_error("operator* dimension mismatch");
152     }
153 }
154
155 // scalar * matrix =====
156 template <typename T1, typename T2>
157 auto operator*(T1 scalar,
158               const std::vector<std::vector<T2>>& inputMatrix)
159 {
160     std::vector<std::vector<T2>> temp {inputMatrix};
161     for(int i = 0; i<inputMatrix.size(); ++i){
162         std::transform(inputMatrix[i].begin(),
163                        inputMatrix[i].end(),
164                        temp[i].begin(),
165                        [&scalar](T2 x){return static_cast<T2>(scalar) * x;});
166     }

```

```

166     return temp;
167 }
168 // // scalar * matrix =====
169 // template <typename T1,
170 //         typename T2,
171 //         typename = typename std::enable_if<std::is_arithmetic<T1>::value>::type>
172 // auto operator*(T1 scalar,
173 //         const std::vector<std::vector<T2>>& inputMatrix)
174 // {
175 //     std::vector<std::vector<T2>> temp {inputMatrix};
176 //     for(int i = 0; i<inputMatrix.size(); ++i){
177 //         std::transform(inputMatrix[i].begin(),
178 //             inputMatrix[i].end(),
179 //             temp[i].begin(),
180 //             [&scalar](T2 x){return static_cast<T2>(scalar) * x;});
181 //     }
182 //     return temp;
183 // }
184 // matrix-multiplication =====
185 template <typename T1, typename T2>
186 auto matmul(const std::vector<std::vector<T1>>& matA,
187             const std::vector<std::vector<T2>>& matB)
188 {
189     // throwing error
190     if (matA[0].size() != matB.size()) {std::cerr << "dimension-mismatch \n";}
191
192     // getting result-type
193     using ResultType = decltype(std::declval<T1>() * std::declval<T2>() + \
194                                std::declval<T1>() * std::declval<T2>());
195
196     // creating aliases
197     auto finalnumrows {matA.size()};
198     auto finalnumcols {matB[0].size()};
199
200     // creating placeholder
201     auto rowcolproduct = [&](auto rowA, auto colB){
202         ResultType temp {0};
203         for(int i = 0; i < matA.size(); ++i) {temp += static_cast<ResultType>(matA[rowA][i]) +
204             static_cast<ResultType>(matB[i][colB]);}
205         return temp;
206     };
207
208     // producing row-column combinations
209     std::vector<std::vector<ResultType>> finaloutput(finalnumrows, std::vector<ResultType>(finalnumcols));
210     for(int row = 0; row < finalnumrows; ++row){for(int col = 0; col < finalnumcols;
211         ++col){finaloutput[row][col] = rowcolproduct(row, col);}}
212
213     // returning
214     return finaloutput;
215 }
216 // matrix * vector =====
217 template <typename T>
218 auto operator*(const std::vector<std::vector<T>> input_matrix,
219               const std::vector<T> input_vector)
220 {
221     // fetching dimensions
222     const auto& num_rows_matrix {input_matrix.size()};
223     const auto& num_cols_matrix {input_matrix[0].size()};
224     const auto& num_rows_vector {1};
225     const auto& num_cols_vector {input_vector.size()};
226
227     const auto& max_num_rows {num_rows_matrix > num_rows_vector ? \
228         num_rows_matrix : num_rows_vector};
229     const auto& max_num_cols {num_cols_matrix > num_cols_vector ? \
230         num_cols_matrix : num_cols_vector};
231
232     // creating canvas
233     auto canvas {std::vector<std::vector<T>>(
234         max_num_rows,
235         std::vector<T>(max_num_cols, 0)
236     )};
237
238     //
239     if (num_cols_matrix == 1 && num_rows_vector == 1){

```

```

239
240     // writing to canvas
241     for(auto row = 0; row < max_num_rows; ++row)
242         for(auto col = 0; col < max_num_cols; ++col)
243             canvas[row][col] = input_matrix[row][0] * input_vector[col];
244 }
245 else{
246     std::cerr << "Operator*: [matrix, vector] | not implemented \n";
247 }
248
249 // returning
250 return std::move(canvas);
251
252 }
253
254 // scalar operators =====
255 auto operator*(const std::complex<double> complexscalar,
256               const double doublescalar){
257     return complexscalar * static_cast<std::complex<double>>(doublescalar);
258 }
259 auto operator*(const double doublescalar,
260               const std::complex<double> complexscalar){
261     return complexscalar * static_cast<std::complex<double>>(doublescalar);
262 }
263 auto operator*(const std::complex<double> complexscalar,
264               const int scalar){
265     return complexscalar * static_cast<std::complex<double>>(scalar);
266 }
267 auto operator*(const int scalar,
268               const std::complex<double> complexscalar){
269     return complexscalar * static_cast<std::complex<double>>(scalar);
270 }

```

10.24 Subtraction

```

1 // =====
2 // Aim: subtracting scalar from a vector
3 template <typename T>
4 std::vector<T> operator-(const std::vector<T>& a, const T scalar){
5     std::vector<T> temp(a.size());
6     std::transform(a.begin(),
7                   a.end(),
8                   temp.begin(),
9                   [scalar](T x){return (x - scalar);});
10    return temp;
11 }
12 // =====
13 template <typename T>
14 auto operator-(const std::vector<std::vector<T>>& input_matrix_A,
15               const std::vector<std::vector<T>>& input_matrix_B)
16 {
17     // throwing error in case of dimension differences
18     if (input_matrix_A.size() != input_matrix_B.size() ||
19         input_matrix_A[0].size() != input_matrix_B[0].size())
20         std::cerr << "operator- dimension mismatch\n";
21
22     // setting up canvas
23     auto canvas {std::vector<std::vector<T>>>(
24         input_matrix_A.size(),
25         std::vector<T>(input_matrix_A[0].size())
26     )};
27
28     // subtracting values
29     for(auto row = 0; row < input_matrix_B.size(); ++row)
30         std::transform(input_matrix_A[row].begin(), input_matrix_A[row].end(),
31                       input_matrix_B[row].begin(),
32                       canvas[row].begin(),
33                       [](const auto& x, const auto& y){
34                           return x - y;
35                       });

```

```

36
37     // returning
38     return std::move(canvas);
39
40 }

```

10.25 Operator Overloadings

10.26 Printing Containers

```

1  // vector printing function
2  template<typename T>
3  void fPrintVector(vector<T> input){
4      for(auto x: input) cout << x << ", ";
5      cout << endl;
6  }
7
8  template<typename T>
9  void fpv(vector<T> input){
10     for(auto x: input) cout << x << ", ";
11     cout << endl;
12 }
13 // =====
14 template<typename T>
15 void fPrintMatrix(const std::vector<std::vector<T>> input_matrix){
16     for(const auto& row: input_matrix)
17         cout << format("{}\n", row);
18 }
19 template <typename T>
20 void fPrintMatrix(const string& input_string,
21                  const std::vector<std::vector<T>> input_matrix){
22     cout << format("{} = \n", input_string);
23     for(const auto& row: input_matrix)
24         cout << format("{}\n", row);
25 }
26
27
28 template<typename T, typename T1>
29 void fPrintHashmap(unordered_map<T, T1> input){
30     for(auto x: input){
31         cout << format("{}\n", x.first, x.second);
32     }
33     cout << endl;
34 }
35
36 void fPrintBinaryTree(TreeNode* root){
37     // sending it back
38     if (root == nullptr) return;
39
40     // printing
41     PRINTLINE
42     cout << "root->val = " << root->val << endl;
43
44     // calling the children
45     fPrintBinaryTree(root->left);
46     fPrintBinaryTree(root->right);
47
48     // returning
49     return;
50 }
51
52
53 void fPrintLinkedList(ListNode* root){
54     if (root == nullptr) return;
55     cout << root->val << " -> ";

```

```

56     fPrintLinkedList(root->next);
57     return;
58 }
59
60 template<typename T>
61 void fPrintContainer(T input){
62     for(auto x: input) cout << x << ", ";
63     cout << endl;
64     return;
65 }
66 // =====
67 template <typename T>
68 auto size(std::vector<std::vector<T>> inputMatrix){
69     cout << format("{} {},{}\n", inputMatrix.size(), inputMatrix[0].size());
70 }
71
72 template <typename T>
73 auto size(const std::string inputstring, std::vector<std::vector<T>> inputMatrix){
74     cout << format("{} = {},{}\n", inputstring, inputMatrix.size(), inputMatrix[0].size());
75 }

```

10.27 Random Number Generation

```

1 // =====
2 template <typename T>
3 auto rand(const T min, const T max) {
4     static std::random_device rd; // Seed
5     static std::mt19937 gen(rd()); // Mersenne Twister generator
6     std::uniform_real_distribution<> dist(min, max);
7     return dist(gen);
8 }
9 // =====
10 template <typename T>
11 auto rand(const T min,
12           const T max,
13           const size_t numelements)
14 {
15     static std::random_device rd; // Seed
16     static std::mt19937 gen(rd()); // Mersenne Twister generator
17     std::uniform_real_distribution<> dist(min, max);
18
19     // building the finaloutput
20     vector<T> finaloutput(numelements);
21     for(int i = 0; i<finaloutput.size(); ++i) {finaloutput[i] = static_cast<T>(dist(gen));}
22
23     return finaloutput;
24 }
25 // =====
26 template <typename T>
27 auto rand(const T argmin,
28           const T argmax,
29           const vector<int> dimensions)
30 {
31
32     // throwing an error if dimension is greater than two
33     if (dimensions.size() > 2) {std::cerr << "dimensions are too high\n";}
34
35     // creating random engine
36     static std::random_device rd; // Seed
37     static std::mt19937 gen(rd()); // Mersenne Twister generator
38     std::uniform_real_distribution<> dist(argmin, argmax);
39
40     // building the finaloutput
41     vector<vector<T>> finaloutput;
42     for(int i = 0; i<dimensions[0]; ++i){
43         vector<T> temp;
44         for(int j = 0; j<dimensions[1]; ++j) {temp.push_back(dist(gen));}
45         // cout << format("\t\t temp = {}\n", temp);
46
47         finaloutput.push_back(temp);

```

```

48     }
49
50     // returning the finaloutput
51     return finaloutput;
52
53 }
54 // =====
55 auto rand(const vector<int> dimensions)
56 {
57
58     using ReturnType = double;
59
60     // throwing an error if dimension is greater than two
61     if (dimensions.size() > 2) {std::cerr << "dimensions are too high\n";}
62
63     // creating random engine
64     static std::random_device rd; // Seed
65     static std::mt19937 gen(rd()); // Mersenne Twister generator
66     std::uniform_real_distribution<> dist(0.00, 1.00);
67
68     // building the finaloutput
69     vector<vector<ReturnType>> finaloutput;
70     for(int i = 0; i<dimensions[0]; ++i){
71         vector<ReturnType> temp;
72         for(int j = 0; j<dimensions[1]; ++j) {temp.push_back(dist(gen));}
73         finaloutput.push_back(std::move(temp));
74     }
75
76     // returning the finaloutput
77     return std::move(finaloutput);
78
79 }
80 // =====
81 template <typename T>
82 auto rand_complex_double(const T argmin,
83                          const T argmax,
84                          const vector<int>& dimensions)
85 {
86
87     // throwing an error if dimension is greater than two
88     if (dimensions.size() > 2) {std::cerr << "dimensions are too high\n";}
89
90     // creating random engine
91     static std::random_device rd; // Seed
92     static std::mt19937 gen(rd()); // Mersenne Twister generator
93     std::uniform_real_distribution<> dist(argmin, argmax);
94
95     // building the finaloutput
96     vector<vector<complex<double>>> finaloutput;
97     for(int i = 0; i<dimensions[0]; ++i){
98         vector<complex<double>> temp;
99         for(int j = 0; j<dimensions[1]; ++j) {temp.push_back(static_cast<double>(dist(gen)));}
100         finaloutput.push_back(std::move(temp));
101     }
102
103     // returning the finaloutput
104     return finaloutput;
105
106 }

```

10.28 Reshape

```

1 // =====
2 // reshaping a matrix into another matrix
3 template <std::size_t M, std::size_t N, typename T>
4 auto reshape(const std::vector<std::vector<T>>& input_matrix){
5
6     // verifying size stuff
7     if (M*N != input_matrix.size() * input_matrix[0].size())
8         std::cerr << "Dimensions are quite different\n";

```

```

9
10 // creating canvas
11 auto canvas {std::vector<std::vector<T>>>(
12     M, std::vector<T>(N, (T)0)
13 );};
14
15 // writing to canvas
16 size_t tid {0};
17 size_t target_row {0};
18 size_t target_col {0};
19 for(auto row = 0; row<input_matrix.size(); ++row){
20     for(auto col = 0; col < input_matrix[0].size(); ++col){
21         tid = row * input_matrix[0].size() + col;
22         target_row = tid/N;
23         target_col = tid%N;
24         canvas[target_row][target_col] = input_matrix[row][col];
25     }
26 }
27
28 // moving it back
29 return std::move(canvas);
30 }
31 // =====
32 // reshaping a matrix into a vector
33 template<std::size_t M, typename T>
34 auto reshape(const std::vector<std::vector<T>>& input_matrix){
35
36     // checking element-count validity
37     if (M != input_matrix.size() * input_matrix[0].size())
38         std::cerr << "Number of elements differ\n";
39
40     // creating canvas
41     auto canvas {std::vector<T>(M, 0)};
42
43     // filling canvas
44     for(auto row = 0; row < input_matrix.size(); ++row)
45         for(auto col = 0; col < input_matrix[0].size(); ++col)
46             canvas[row * input_matrix.size() + col] = input_matrix[row][col];
47
48     // moving it back
49     return std::move(canvas);
50 }
51
52 // =====
53 // Matrix to matrix
54 // =====
55 template<typename T>
56 auto reshape(const std::vector<std::vector<T>>& input_matrix,
57             const std::size_t M,
58             const std::size_t N){
59
60     // checking element-count validity
61     if (M * N != input_matrix.size() * input_matrix[0].size())
62         std::cerr << "Number of elements differ\n";
63
64     // creating canvas
65     auto canvas {std::vector<std::vector<T>>>(
66         M, std::vector<T>(N, (T)0)
67 );};
68
69     // writing to canvas
70     size_t tid {0};
71     size_t target_row {0};
72     size_t target_col {0};
73     for(auto row = 0; row<input_matrix.size(); ++row){
74         for(auto col = 0; col < input_matrix[0].size(); ++col){
75             tid = row * input_matrix[0].size() + col;
76             target_row = tid/N;
77             target_col = tid%N;
78             canvas[target_row][target_col] = input_matrix[row][col];
79         }
80     }
81
82     // moving it back
83     return std::move(canvas);

```

```

84 }
85
86 // =====
87 // converting a matrix into a vector
88 // =====
89 template<typename T>
90 auto reshape(const std::vector<std::vector<T>>& input_matrix,
91             const size_t M){
92
93     // checking element-count validity
94     if (M != input_matrix.size() * input_matrix[0].size())
95         std::cerr << "Number of elements differ\n";
96
97     // creating canvas
98     auto canvas {std::vector<T>(M, 0)};
99
100    // filling canvas
101    for(auto row = 0; row < input_matrix.size(); ++row)
102        for(auto col = 0; col < input_matrix[0].size(); ++col)
103            canvas[row * input_matrix.size() + col] = input_matrix[row][col];
104
105    // moving it back
106    return std::move(canvas);
107 }

```

10.29 Summing with containers

```

1 // =====
2 template <std::size_t axis, typename T>
3 auto sum(const std::vector<T>& input_vector) -> std::enable_if_t<axis == 0, std::vector<T>>
4 {
5     // returning the input as is
6     return input_vector;
7 }
8 // =====
9 template <std::size_t axis, typename T>
10 auto sum(const std::vector<std::vector<T>>& input_matrix) -> std::enable_if_t<axis == 0, std::vector<T>>
11 {
12     // creating canvas
13     auto canvas {std::vector<T>(input_matrix[0].size(), 0)};
14
15     // filling up the canvas
16     for(auto row = 0; row < input_matrix.size(); ++row)
17         std::transform(input_matrix[row].begin(), input_matrix[row].end(),
18                        canvas.begin(),
19                        canvas.begin(),
20                        [](auto& argx, auto& argy){return argx + argy;});
21
22     // returning
23     return std::move(canvas);
24 }
25 }
26 // =====
27 template <std::size_t axis, typename T>
28 auto sum(const std::vector<std::vector<T>>& input_matrix) -> std::enable_if_t<axis == 1,
29             std::vector<std::vector<T>>>
30 {
31     // creating canvas
32     auto canvas {std::vector<std::vector<T>>(input_matrix.size(),
33                                             std::vector<T>(1, 0.00))};
34
35     // filling up the canvas
36     for(auto row = 0; row < input_matrix.size(); ++row)
37         canvas[row][0] = std::accumulate(input_matrix[row].begin(),
38                                           input_matrix[row].end(),
39                                           static_cast<T>(0));
40
41     // returning
42     return std::move(canvas);

```



```

43 }
44 // =====
45 template <std::size_t axis, typename T>
46 auto sum(const std::vector<T>& input_vector_A,
47         const std::vector<T>& input_vector_B) -> std::enable_if_t<axis == 0, std::vector<T> >
48 {
49     // setup
50     const auto& num_cols_A {input_vector_A.size()};
51     const auto& num_cols_B {input_vector_B.size()};
52
53     // throwing errors
54     if (num_cols_A != num_cols_B) {std::cerr << "sum: size disparity\n";}
55
56     // creating canvas
57     auto canvas {input_vector_A};
58
59     // summing up
60     std::transform(input_vector_B.begin(), input_vector_B.end(),
61                  canvas.begin(),
62                  canvas.begin(),
63                  std::plus<T>());
64
65     // returning
66     return std::move(canvas);
67 }

```

10.30 Tangent

```

1 namespace svr {
2     // =====
3     template <typename T>
4     auto atan2(const std::vector<T> input_vector_A,
5              const std::vector<T> input_vector_B)
6     {
7         // throw error
8         if (input_vector_A.size() != input_vector_B.size())
9             std::cerr << "atan2: size disparity\n";
10
11         // create canvas
12         auto canvas {std::vector<T>(input_vector_A.size(), 0)};
13
14         // performing element-wise atan2 calculation
15         std::transform(input_vector_A.begin(), input_vector_A.end(),
16                      input_vector_B.begin(),
17                      canvas.begin(),
18                      [](const auto& arg_a,
19                       const auto& arg_b){
20
21                          return std::atan2(arg_a, arg_b);
22                      });
23
24         // moving things back
25         return std::move(canvas);
26     }
27     // =====
28     template <typename T>
29     auto atan2(T scalar_A,
30              T scalar_B)
31     {
32         return std::atan2(scalar_A, scalar_B);
33     }
34 }

```

10.31 Tiling Operations

```

1 namespace svr {

```

```

2 // =====
3 template <typename T>
4 auto tile(const std::vector<T>& input_vector,
5           const std::vector<size_t> mul_dimensions){
6
7     // creating canvas
8     const std::size_t num_rows {1 * mul_dimensions[0]};
9     const std::size_t num_cols {input_vector.size() * mul_dimensions[1]};
10    auto canvas {std::vector<std::vector<T>>>(
11        num_rows,
12        std::vector<T>(num_cols, 0)
13    )};
14
15    // writing
16    std::size_t source_row;
17    std::size_t source_col;
18
19    for(std::size_t row = 0; row < num_rows; ++row){
20        for(std::size_t col = 0; col < num_cols; ++col){
21            source_row = row % 1;
22            source_col = col % input_vector.size();
23            canvas[row][col] = input_vector[source_col];
24        }
25    }
26
27    // returning
28    return std::move(canvas);
29 }
30 // =====
31 template <typename T>
32 auto tile(const std::vector<std::vector<T>>& input_matrix,
33           const std::vector<size_t> mul_dimensions){
34
35     // creating canvas
36     const std::size_t num_rows {input_matrix.size() * mul_dimensions[0]};
37     const std::size_t num_cols {input_matrix[0].size() * mul_dimensions[1]};
38     auto canvas {std::vector<std::vector<T>>>(
39        num_rows,
40        std::vector<T>(num_cols, 0)
41    )};
42
43    // writing
44    std::size_t source_row;
45    std::size_t source_col;
46
47    for(std::size_t row = 0; row < num_rows; ++row){
48        for(std::size_t col = 0; col < num_cols; ++col){
49            source_row = row % input_matrix.size();
50            source_col = col % input_matrix[0].size();
51            canvas[row][col] = input_matrix[source_row][source_col];
52        }
53    }
54
55    // returning
56    return std::move(canvas);
57 }
58 }

```

10.32 Transpose

```

1 template <typename T>
2 auto transpose(const std::vector<T> input_vector){
3
4     // creating canvas
5     auto canvas {std::vector<std::vector<T>>>(
6         input_vector.size(),
7         std::vector<T>(1)
8     )};
9
10    // filling canvas

```

```
11     for(auto i = 0; i < input_vector.size(); ++i){
12         canvas[i][0] = input_vector[i];
13     }
14
15     // moving it back
16     return std::move(canvas);
17 }
```
