

Autonomous Underwater Vehicle: A Surveillance Protocol

S.V. Rajendran

September 29, 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 we will be writing templated STL code.

Contents

Preface	i
I AUV Components & Setup	1
1 Underwater Environment	2
1.1 Underwater Hills	2
1.2 Scatterer Definition	3
1.3 Sea-Floor Setup Script	4
2 Transmitter	6
2.1 Transmission Signal	7
2.2 Transmitter Class Definition	8
2.3 Transmitter Setup Scripts	9
3 Uniform Linear Array	13
3.1 ULA Class Definition	14
3.2 ULA Setup Scripts	16
4 Autonomous Underwater Vehicle	18
4.1 AUV Class Definition	19
4.2 AUV Setup Scripts	20
II Signal Simulation Pipeline	21
III Imaging Pipeline	22
IV Control Pipeline	23
A Software	24
A.1 Class Definitions	24

A.1.1	Class: Scatter	24
A.1.2	Class: Transmitter	26
A.1.3	Class: Uniform Linear Array	38
A.1.4	Class: Autonomous Underwater Vehicle	64
A.2	Setup Scripts	80
A.2.1	Seafloor Setup	80
A.2.2	Transmitter Setup	82
A.2.3	ULA Setup	85
A.2.4	AUV Setup	87
A.3	Function Definitions	88
A.3.1	Cartesian Coordinates to Spherical Coordinates	88
A.3.2	Spherical Coordinates to Cartesian Coordinates	89
A.3.3	Column-Wise Convolution	91
A.3.4	Buffer 2D	93
A.3.5	fAnglesToTensor	94
A.3.6	fCalculateCosine	94
A.4	Main Scripts	96
A.4.1	Signal Simulation	96
B	General Purpose Templated Functions	99
B.1	CSV File-Writes	99
B.2	abs	100
B.3	Boolean Comparators	101
B.4	Concatenate Functions	102
B.5	Conjugate	104
B.6	Convolution	104
B.7	Coordinate Change	105
B.8	Cosine	107
B.9	Data Structures	108
B.10	Editing Index Values	108
B.11	Equality	109
B.12	Exponentiate	109
B.13	FFT	109
B.14	Flipping Containers	111
B.15	Indexing	112
B.16	Linspace	113
B.17	Max	114

B.18	Meshgrid	114
B.19	Minimum	115
B.20	Norm	116
B.21	Division	116
B.22	Addition	117
B.23	Multiplication (Element-wise)	120
B.24	Subtraction	124
B.25	Operator Overloadings	126
B.26	Printing Containers	126
B.27	Random Number Generation	127
B.28	Reshape	129
B.29	Summing with containers	131
B.30	Tangent	132
B.31	Tiling Operations	133
B.32	Transpose	134
B.33	Masking	134
B.34	Resetting Containers	136
B.35	Element-wise squaring	136
B.36	Thread-Pool	138

Part I

AUV Components & Setup

Chapter 1

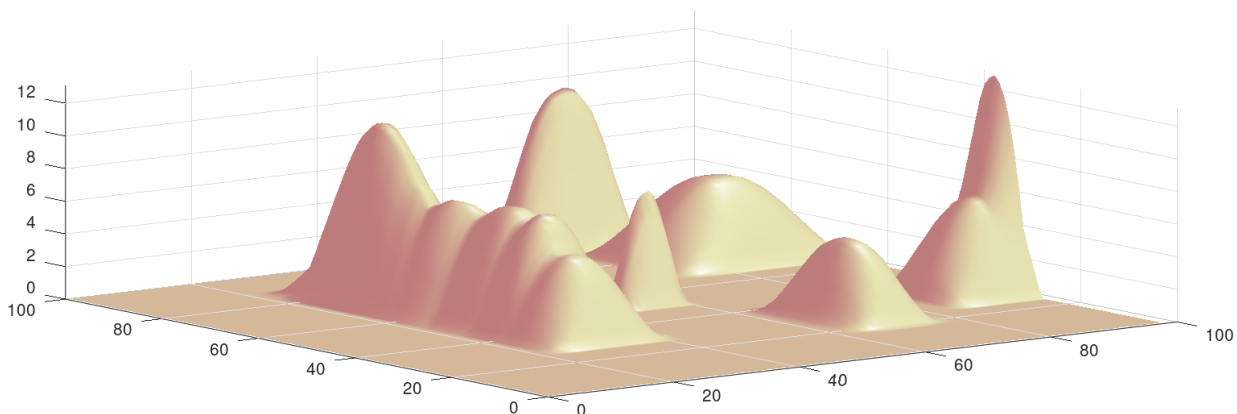
Underwater Environment

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.

To simplify things, we shall take a more constrained and structured approach. We start by creating different classes of structures and produce instantiations of those structures on the sea-floor. These structures are defined in such a way that the shape and size can be parameterized to enable creation of random sea-floors.



1.1 Underwater Hills

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 1. 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:  $\mathbf{P} \leftarrow \text{concatenate}([\mathbf{P}, H])$ 

```

1.2 Scatterer Definition

The sea-floor is represented by a single object of the class ScattererClass.

```

1  /*=====
2  Class Declaration
3  -----*/
4  template <typename T>
5  class ScattererClass
6  {
7  public:
8      // members
9      std::vector<std::vector<T>> coordinates;
10     std::vector<T> reflectivity;
11
12     // Constructor
13     ScattererClass() {}
14
15     // Constructor
16     ScattererClass(std::vector<std::vector<T>> coordinates_arg,
17                    std::vector<T> reflectivity_arg):
18         coordinates(std::move(coordinates_arg)),
19         reflectivity(std::move(reflectivity_arg)) {}
20
21     // Save to CSV

```



```

22     void save_to_csv();
23 };

```

1.3 Sea-Floor Setup Script

Following is the function that will setup the sea-floor script.

```

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 -
58                               hill_dimensions_min)}; // 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)};
67
68     // populating the values of the incoming argument
69     scatterers.coordinates = std::move(points2D);
70     scatterers.reflectivity = std::move(floorScatter_reflectivity);
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 }

```

Chapter 2

Transmitter

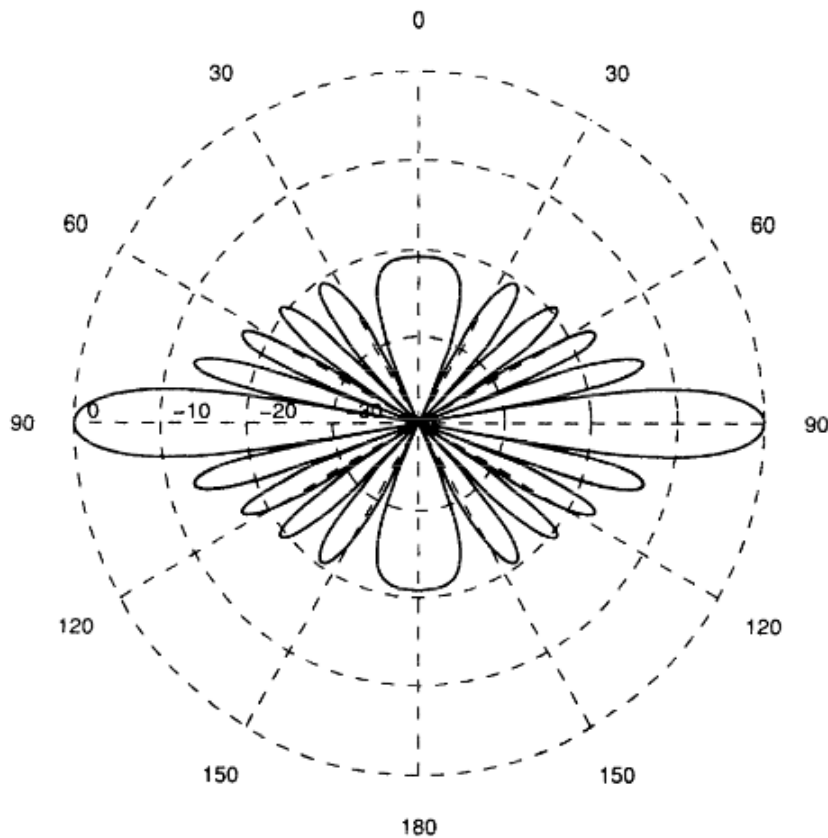


Figure 2.1: Beampattern of a Transmission Uniform Linear Array

Overview

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.

A transmitter is any device or circuit that converts information into a signal and sends it out onto some media like air, cable, water or space. The components of a transmitter are usually as follows

1. Input: Information containing signal such as voice, data, video etc
2. Process: Encode/modulate the information onto a carrier signal, which can be electromagnetic wave or mechanical wave.
3. Transmission: The signal is then transmitted onto the media with electro-mechanical equipment.

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.

2.1 Transmission Signal

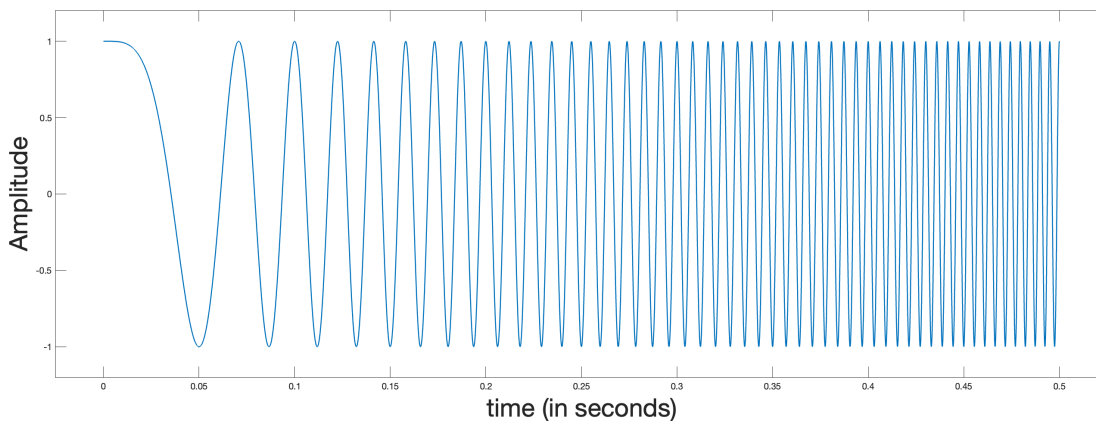


Figure 2.2: Linear Frequency Modulated Wave

The resolution of any probing system is fundamentally tied to the signal bandwidth. A higher bandwidth corresponds to finer resolution $\frac{\text{speed-of-sounds}}{2 \cdot \text{bandwidth}}$. Thus, for perfect resolution, an infinite bandwidth is in order. However, infinite bandwidth is impossible for obvious reasons: hardware limitations, spectral regulations, energy limitations and so on.

This is where Linear Frequency Modulation (LFM), also called a “chirp,” becomes valuable. An LFM signal linearly sweeps a limited bandwidth over a relatively long duration. This technique spreads the signal’s energy in time while retaining the resolution benefits of

the bandwidth. After matched filtering (or pulse compression), we essentially produce pulses corresponding to a base-band LFM of same bandwidth. Overall, LFM is a practical compromise between finite bandwidth and desired performance.

One of the best parts about the resolution depending only on the bandwidth is that it allows us to deploy techniques that would help us improve SNRs without virtually increasing the bandwidth at all. Much of the noise in submarine environments are in and around the baseband region (around frequency, 0). Since resolution depends purely on bandwidth, and LFM can be transmitted at a carrier-frequency, this means that processing the returns after low-pass filtering and basebanding allows us to get rid of the submarine noise, since they do not occupy the same frequency-coefficients. The end-result, thus, is improved SNR compared to use baseband LFM.

Due to all of these advantages, LFM waves are ubiquitous in probing systems, from sonar to radar. Thus, for this project too, the transmitter will be using LFM waves as probing signals, to probe the surrounding submarine environment.

2.2 Transmitter Class Definition

The transmitter is represented by a single object of the class TransmitterClass.

```

1  template <typename T>
2  class TransmitterClass{
3  public:
4
5      // physical/intrinsic properties
6      std::vector<T>    location;           // location tensor
7      std::vector<T>    pointing_direction; // pointing direction
8
9      // basic parameters
10     std::vector<T>    Signal;             // transmitted signal (LFM)
11     T                  azimuthal_angle;    // transmitter's azimuthal pointing direction
12     T                  elevation_angle;    // transmitter's elevation pointing direction
13     T                  azimuthal_beamwidth; // azimuthal beamwidth of transmitter
14     T                  elevation_beamwidth; // elevation beamwidth of transmitter
15     T                  range;              // a parameter used for spotlight mode.
16
17     // transmitted signal attributes
18     T                  f_low;              // lowest frequency of LFM
19     T                  f_high;             // highest frequency of LFM
20     T                  fc;                 // center frequency of LFM
21     T                  bandwidth;          // bandwidth of LFM
22
23     // shadowing properties
24     int                azimuthQuantDensity; // quantization of angles along the
        azimuth
25     int                elevationQuantDensity; // quantization of angles along the
        elevation
26     T                  rangeQuantSize;      // range-cell size when shadowing
27     T                  azimuthShadowThreshold; // azimuth thresholding
28     T                  elevationShadowThreshold; // elevation thresholding
29
30     // shadowing related
31     std::vector<T>    checkbox;             // box indicating whether a scatter for a
        range-angle pair has been found

```

```

32     std::vector<std::vector<std::vector<T>>> finalScatterBox; // a 3D tensor where the
        third dimension represnets the vector length
33     std::vector<T> finalReflectivityBox; // to store the reflectivity
34
35     // constructor
36     TransmitterClass() = default;
37
38     // Deleting copy constructors/assignment
39     TransmitterClass(const TransmitterClass& other)      = delete;
40     TransmitterClass& operator=(TransmitterClass& other) = delete;
41
42     // Creating move-constructor and move-assignment
43     TransmitterClass(TransmitterClass&& other)          = default;
44     TransmitterClass& operator=(TransmitterClass&& other) = default;
45
46     // member-functions
47     auto updatePointingAngle(std::vector<T> AUV_pointing_vector);
48     auto subset_scatterers(const ScattererClass<T>& seafloor,
49                           std::vector<std::size_t>& indices,
50                           const T& tilt_angle);
51
52 };

```

2.3 Transmitter Setup Scripts

The following script shows the setup-script

```

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
22                                           = 0
23
24     // Setting up transmitter
25     auto  location              {std::vector<T>(3, 0)}; // location of
        transmitter
26     T    azimuthal_angle_fls   {0};                // initial
        pointing direction
27     T    azimuthal_angle_port  {90};               // initial

```

```

27     pointing direction
T     azimuthal_angle_starboard    {-90};           // initial
    pointing direction
28
29     elevation_angle
T     elevation_angle              {-60};           // initial
    pointing direction
30
31     azimuthal_beamwidth_fls
T     azimuthal_beamwidth_fls      {20};           // azimuthal
    beamwidth of the signal cone
32     azimuthal_beamwidth_port
T     azimuthal_beamwidth_port      {20};           // azimuthal
    beamwidth of the signal cone
33     azimuthal_beamwidth_starboard
T     azimuthal_beamwidth_starboard {20};           // azimuthal
    beamwidth of the signal cone
34
35     elevation_beamwidth_fls
T     elevation_beamwidth_fls       {20};           // elevation
    beamwidth of the signal cone
36     elevation_beamwidth_port
T     elevation_beamwidth_port       {20};           // elevation
    beamwidth of the signal cone
37     elevation_beamwidth_starboard
T     elevation_beamwidth_starboard  {20};           // elevation
    beamwidth of the signal cone
38
39     int    azimuthQuantDensity      {10}; // number of points, a degree is split
    into quantization density along azimuth (used for shadowing)
40     int    elevationQuantDensity    {10}; // number of points, a degree is split
    into quantization density along elevation (used for shadowing)
41     T      rangeQuantSize           {10}; // the length of a cell (used for
    shadowing)
42
43     T      azimuthShadowThreshold    {1}; // azimuth threshold (in degrees)
44     T      elevationShadowThreshold  {1}; // elevation threshold (in degrees)
45
46
47 // transmitter-fls
48 transmitter_fls.location            = location;           // Assigning
    location
49 transmitter_fls.Signal              = Signal;             // Assigning
    signal
50 transmitter_fls.azimuthal_angle      = azimuthal_angle_fls; // assigning
    azimuth angle
51 transmitter_fls.elevation_angle      = elevation_angle;    // assigning
    elevation angle
52 transmitter_fls.azimuthal_beamwidth  = azimuthal_beamwidth_fls; // assigning
    azimuth-beamwidth
53 transmitter_fls.elevation_beamwidth  = elevation_beamwidth_fls; // assigning
    elevation-beamwidth
54 // updating quantization densities
55 transmitter_fls.azimuthQuantDensity  = azimuthQuantDensity; // assigning
    azimuth quant density
56 transmitter_fls.elevationQuantDensity = elevationQuantDensity; // assigning
    elevation quant density
57 transmitter_fls.rangeQuantSize        = rangeQuantSize;    // assigning
    range-quantization
58 transmitter_fls.azimuthShadowThreshold = azimuthShadowThreshold; //
    azimuth-threshold in shadowing
59 transmitter_fls.elevationShadowThreshold = elevationShadowThreshold; //
    elevation-threshold in shadowing
60 // signal related
61 transmitter_fls.f_low                = f1;                // assigning lower frequency
62 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
69   location
70 transmitter_portside.Signal        = Signal;       // Assigning
71   signal
72 transmitter_portside.azimuthal_angle = azimuthal_angle_port; // assigning
73   azimuth angle
74 transmitter_portside.elevation_angle = elevation_angle; // assigning
75   elevation angle
76 transmitter_portside.azimuthal_beamwidth = azimuthal_beamwidth_port; // assigning
77   azimuth-beamwidth
78 transmitter_portside.elevation_beamwidth = elevation_beamwidth_port; // assigning
79   elevation-beamwidth
80 // updating quantization densities
81 transmitter_portside.azimuthQuantDensity = azimuthQuantDensity; // assigning
82   azimuth quant density
83 transmitter_portside.elevationQuantDensity = elevationQuantDensity; // assigning
84   elevation quant density
85 transmitter_portside.rangeQuantSize      = rangeQuantSize; // assigning
86   range-quantization
87 transmitter_portside.azimuthShadowThreshold = azimuthShadowThreshold; //
88   azimuth-threshold in shadowing
89 transmitter_portside.elevationShadowThreshold = elevationShadowThreshold; //
90   elevation-threshold in shadowing
91 // signal related
92 transmitter_portside.f_low              = f1;       // assigning
93   lower frequency
94 transmitter_portside.f_high             = f2;       // assigning
95   higher frequency
96 transmitter_portside.fc                 = fc;       // assigning
97   center frequency
98 transmitter_portside.bandwidth          = bandwidth; // assigning
99   bandwidth
100
101
102 // transmitter-starboard
103 transmitter_starboard.location           = location; //
104   assigning location
105 transmitter_starboard.Signal             = Signal;   //
106   assigning signal
107 transmitter_starboard.azimuthal_angle     = azimuthal_angle_starboard; //
108   assigning azimuthal signal
109 transmitter_starboard.elevation_angle     = elevation_angle;
110 transmitter_starboard.azimuthal_beamwidth = azimuthal_beamwidth_starboard;
111 transmitter_starboard.elevation_beamwidth = elevation_beamwidth_starboard;
112 // updating quantization densities
113 transmitter_starboard.azimuthQuantDensity = azimuthQuantDensity; //
114   assigning azimuth-quant-density
115 transmitter_starboard.elevationQuantDensity = elevationQuantDensity;
116 transmitter_starboard.rangeQuantSize      = rangeQuantSize;
117 transmitter_starboard.azimuthShadowThreshold = azimuthShadowThreshold;
118 transmitter_starboard.elevationShadowThreshold = elevationShadowThreshold;
119 // signal related
120 transmitter_starboard.f_low               = f1;       //
121   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 }
```

Chapter 3

Uniform Linear Array

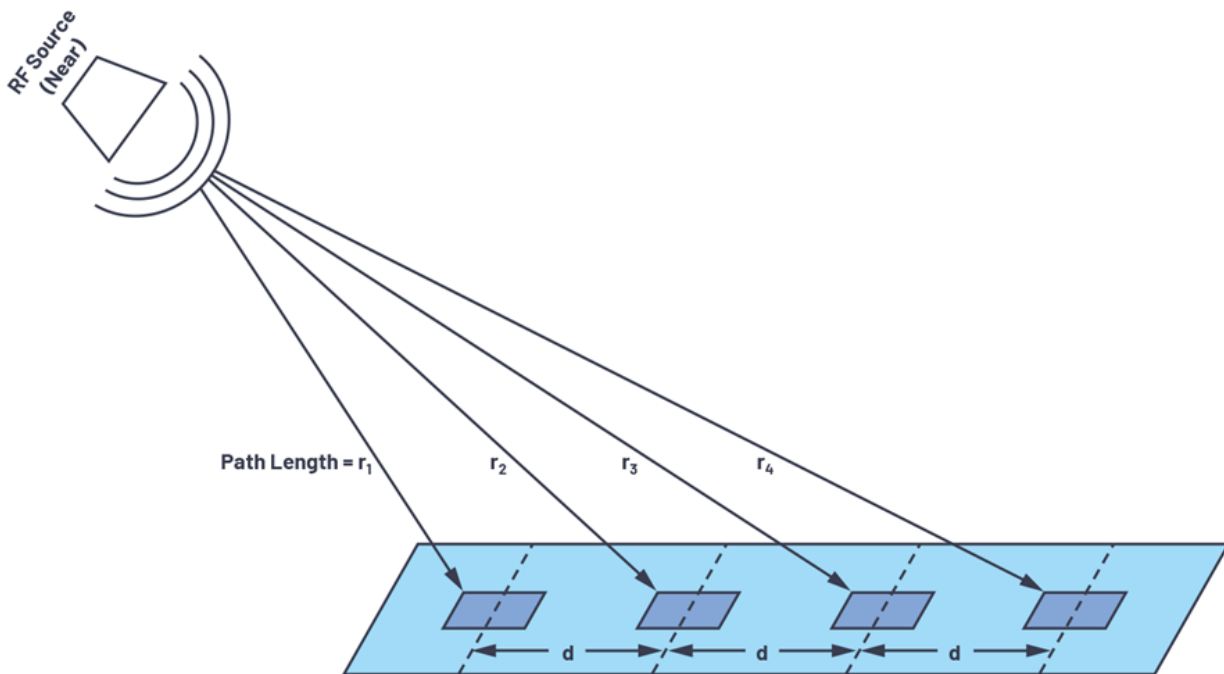


Figure 3.1: Uniform Linear Array

Overview

A Uniform Linear Array (ULA) is a common antenna or sensor configuration in which multiple elements are arranged in a straight line with equal spacing between adjacent elements. This geometry simplifies both the analysis and implementation of array signal processing techniques. In a ULA, each element receives a version of the incoming signal that differs only in phase, depending on the angle of arrival. This phase difference can be exploited to steer the array's beam in a desired direction (beamforming) or to estimate the direction of arrival (DOA) of multiple sources. The equal spacing also leads to a regular phase progression across the elements, which makes the array's response mathematically tractable and allows the use of tools like the discrete Fourier transform (DFT) to analyze spatial frequency content.

The performance of a ULA depends on the number of elements and their spacing. The spacing is typically chosen to be half the wavelength of the signal to avoid spatial aliasing, also called grating lobes, which can introduce ambiguities in DOA estimation. Increasing the number of elements improves the array's angular resolution and directivity, meaning it can better distinguish closely spaced sources and focus energy more narrowly. ULAs are widely used in radar, sonar, wireless communications, and microphone arrays due to their simplicity, predictable behavior, and compatibility with well-established signal processing algorithms. Their linear structure also makes them easier to implement in hardware compared to more complex array geometries like circular or planar arrays.

3.1 ULA Class Definition

The following is the class used to represent the uniform linear array

```

1  template <typename T>
2  class ULAClass
3  {
4  public:
5      // intrinsic parameters
6      int          num_sensors;                // number of
          sensors
7      T            inter_element_spacing;      // space between
          sensors
8      std::vector<std::vector<T>> coordinates;  // coordinates
          of each sensor
9      T            sampling_frequency;         // sampling
          frequency of the sensors
10     T            recording_period;           // recording
          period of the ULA
11     std::vector<T> location;                 // location of
          first coordinate
12
13     // derived
14     std::vector<T> sensor_direction;
15     std::vector<std::vector<T>> signalMatrix;
16
17     // decimation related
18     int          decimation_factor;           // the new decimation
          factor
19     T            post_decimation_sampling_frequency; // the new sampling
          frequency
20     std::vector<T> lowpass_filter_coefficients_for_decimation; // filter-coefficients
          for filtering
21
22     // imaging related
23     T range_resolution;                      // theoretical range-resolution =  $\frac{c}{2B}$ 
24     T azimuthal_resolution;                  // theoretical azimuth-resolution =
           $\frac{\lambda}{(N-1) \cdot \text{inter-element-distance}}$ 
25     T range_cell_size;                      // the range-cell quanta we're choosing for
          efficiency trade-off
26     T azimuth_cell_size;                    // the azimuth quanta we're choosing
27     std::vector<T> azimuth_centers;          // tensor containing the azimuth centers
28     std::vector<T> range_centers;           // tensor containing the range-centers
29     int frame_size;                         // the frame-size corresponding to a range cell in a
          decimated signal matrix
30

```

```

31     std::vector<std::vector<complex<T>>> mulFFTMMatrix; // the matrix containing the
        delays for each-element as a slot
32     std::vector<complex<T>> matchFilter; // torch tensor containing the
        match-filter
33     int num_buffer_zeros_per_frame; // number of zeros we're adding
        per frame to ensure no-rotation
34     std::vector<std::vector<T>> beamformedImage; // the beamformed image
35     std::vector<std::vector<T>> cartesianImage; // the cartesian version of
        beamformed image
36
37     // Artificial acoustic-image related
38     std::vector<std::vector<T>> currentArtificialAcousticImage; // acoustic image
        directly produced
39
40
41     // Basic Constructor
42     ULAClass() = default;
43
44     // constructor
45     ULAClass(const int num_sensors_arg,
46               const auto inter_element_spacing_arg,
47               const auto& coordinates_arg,
48               const auto& sampling_frequency_arg,
49               const auto& recording_period_arg,
50               const auto& location_arg,
51               const auto& signalMatrix_arg,
52               const auto& lowpass_filter_coefficients_for_decimation_arg):
53         num_sensors(num_sensors_arg),
54         inter_element_spacing(inter_element_spacing_arg),
55         coordinates(std::move(coordinates_arg)),
56         sampling_frequency(sampling_frequency_arg),
57         recording_period(recording_period_arg),
58         location(std::move(location_arg)),
59         signalMatrix(std::move(signalMatrix_arg)),
60         lowpass_filter_coefficients_for_decimation(std::move(lowpass_filter_coefficients_for_decima
61     {
62
63         // calculating ULA direction
64         sensor_direction = std::vector<T>{coordinates[1][0] - coordinates[0][0],
65                                           coordinates[1][1] - coordinates[0][1],
66                                           coordinates[1][2] - coordinates[0][2]};
67
68         // normalizing
69         auto norm_value_temp {std::inner_product(sensor_direction.begin(),
70                                                   sensor_direction.end(),
71                                                   sensor_direction.begin(),
72                                                   0.00)};
73
74         // dividing
75         if (norm_value_temp != 0) {sensor_direction = sensor_direction /
76             norm_value_temp;}
77
78     }
79
80     // deleting copy constructor/assignment
81     // ULAClass(const ULAClass& other) = delete;
82     // ULAClass& operator=(const ULAClass& other) = delete;

```

```

83 // member-functions
84 void buildCoordinatesBasedOnLocation();
85 void init(const TransmitterClass<T>& transmitterObj);
86 void nfdc_CreateMatchFilter(const TransmitterClass<T>& transmitterObj);
87 void simulate_signals(const ScattererClass<T>& seafloor,
88                      const std::vector<std::size_t> scatterer_indices,
89                      const TransmitterClass<T>& transmitter);
90
91 };

```

3.2 ULA Setup Scripts

The following script shows the setup-script for Uniform Linear Arrays

```

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
10     samples
11     T recording_period         {10e-2};                        // sampling-period
12
13     // building the direction for the sensors
14     auto ULA_direction         {std::vector<T>({-1, 0, 0})};
15     auto ULA_direction_norm     {norm(ULA_direction)};
16     if (ULA_direction_norm != 0) {ULA_direction = ULA_direction/ULA_direction_norm;}
17     ULA_direction              = ULA_direction * inter_element_spacing;
18
19     // building coordinates for sensors
20     auto ULA_coordinates        {transpose(ULA_direction) * \
21                                 linspace<double>(0.00,
22                                                    num_sensors -1,
23                                                    num_sensors)};
24
25     // coefficients of decimation filter
26     auto lowpassfiltercoefficients {std::vector<T>{0.0000, 0.0000, 0.0000, 0.0000,
27     0.0000, 0.0000, 0.0001, 0.0003, 0.0006, 0.0015, 0.0030, 0.0057, 0.0100, 0.0163,
28     0.0251, 0.0364, 0.0501, 0.0654, 0.0814, 0.0966, 0.1093, 0.1180, 0.1212, 0.1179,
29     0.1078, 0.0914, 0.0699, 0.0451, 0.0192, -0.0053, -0.0262, -0.0416, -0.0504,
30     -0.0522, -0.0475, -0.0375, -0.0239, -0.0088, 0.0057, 0.0179, 0.0263, 0.0303,
31     0.0298, 0.0253, 0.0177, 0.0086, -0.0008, -0.0091, -0.0153, -0.0187, -0.0191,
32     -0.0168, -0.0123, -0.0065, -0.0004, 0.0052, 0.0095, 0.0119, 0.0125, 0.0112,
33     0.0084, 0.0046, 0.0006, -0.0031, -0.0060, -0.0078, -0.0082, -0.0075, -0.0057,
34     -0.0033, -0.0006, 0.0019, 0.0039, 0.0051, 0.0055, 0.0050, 0.0039, 0.0023, 0.0005,
35     -0.0012, -0.0025, -0.0034, -0.0036, -0.0034, -0.0026, -0.0016, -0.0004, 0.0007,
36     0.0016, 0.0022, 0.0024, 0.0023, 0.0018, 0.0011, 0.0003, -0.0004, -0.0011,
37     -0.0015, -0.0016, -0.0015}};
38
39     // assigning values
40     ula_fls.num_sensors          = num_sensors;                //
41     ula_fls.inter_element_spacing = inter_element_spacing;    //

```

```

    assigning inter-element spacing
30  ula_fls.coordinates = ULA_coordinates; //
    assigning ULA coordinates
31  ula_fls.sampling_frequency = sampling_frequency; //
    assigning sampling frequencys
32  ula_fls.recording_period = recording_period; //
    assigning recording period
33  ula_fls.sensor_direction = ULA_direction; // ULA
    direction
34  ula_fls.lowpass_filter_coefficients_for_decimation = lowpassfiltercoefficients; //
    storing coefficients
35
36
37  // assigning values
38  ula_portside.num_sensors = num_sensors; //
    assigning number of sensors
39  ula_portside.inter_element_spacing = inter_element_spacing; //
    assigning inter-element spacing
40  ula_portside.coordinates = ULA_coordinates; //
    assigning ULA coordinates
41  ula_portside.sampling_frequency = sampling_frequency; //
    assigning sampling frequencys
42  ula_portside.recording_period = recording_period; //
    assigning recording period
43  ula_portside.sensor_direction = ULA_direction; //
    ULA direction
44  ula_portside.lowpass_filter_coefficients_for_decimation = lowpassfiltercoefficients;
    // storing coefficients
45
46
47  // assigning values
48  ula_starboard.num_sensors = num_sensors; //
    assigning number of sensors
49  ula_starboard.inter_element_spacing = inter_element_spacing; //
    assigning inter-element spacing
50  ula_starboard.coordinates = ULA_coordinates; //
    assigning ULA coordinates
51  ula_starboard.sampling_frequency = sampling_frequency; //
    assigning sampling frequencys
52  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 }

```

Chapter 4

Autonomous Underwater Vehicle

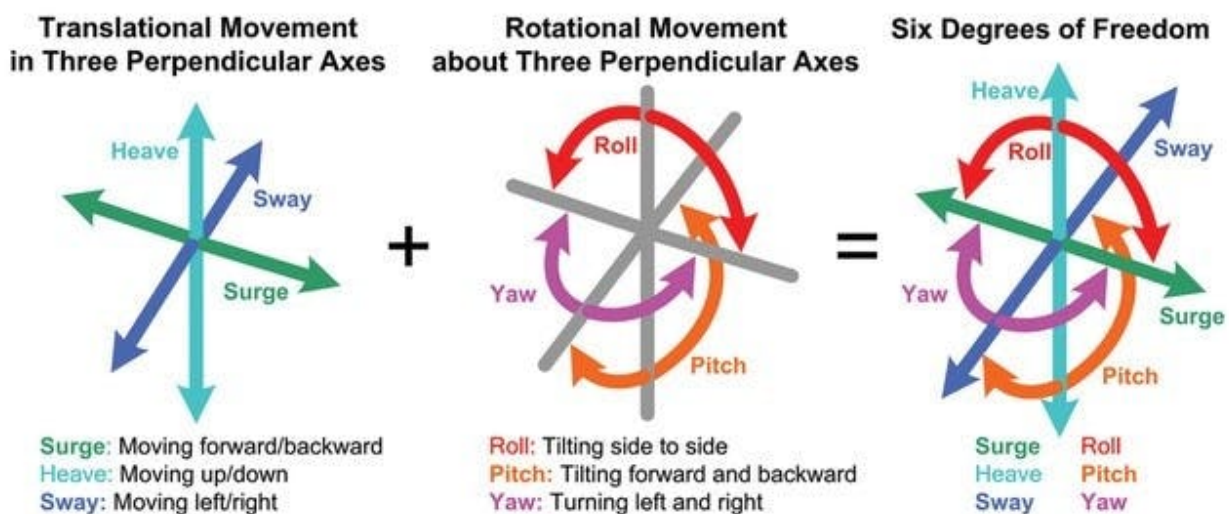


Figure 4.1: AUV degrees of freedom

Overview

Autonomous Underwater Vehicles (AUVs) are robotic systems designed to operate underwater without direct human control. They navigate and perform missions independently using onboard sensors, processors, and preprogrammed instructions. They are widely used in oceanographic research, environmental monitoring, offshore engineering, and military applications. AUVs can vary in size from small, portable vehicles for shallow water surveys to large, torpedo-shaped platforms capable of deep-sea exploration. Their autonomy allows them to access environments that are too dangerous, remote, or impractical for human divers or tethered vehicles.

The navigation and sensing systems of AUVs are critical to their performance. They typically use a combination of inertial measurement units (IMUs), Doppler velocity logs

(DVLs), pressure sensors, magnetometers, and sometimes acoustic positioning systems to estimate their position and orientation underwater. Since GPS signals do not penetrate water, AUVs must rely on these onboard sensors and occasional surfacing for GPS fixes. They are often equipped with sonar systems, cameras, or other scientific instruments to collect data about the seafloor, water column, or underwater structures. Advanced AUVs can also implement adaptive mission planning and obstacle avoidance, enabling them to respond to changes in the environment in real time.

The applications of AUVs are diverse and expanding rapidly. In scientific research, they are used for mapping the seafloor, studying marine life, and monitoring oceanographic parameters such as temperature, salinity, and currents. In the commercial sector, AUVs inspect pipelines, subsea infrastructure, and offshore oil platforms. Military and defense applications include mine countermeasure operations and underwater surveillance. The development of AUVs continues to focus on increasing endurance, improving autonomy, enhancing sensor payloads, and reducing costs, making them a key technology for exploring and understanding the underwater environment efficiently and safely.

4.1 AUV Class Definition

The following is the class used to represent the uniform linear array

```

1  template <typename T>
2  class  AUVClass{
3  public:
4
5      // Intrinsic attributes
6      std::vector<T>    location;           // location of vessel
7      std::vector<T>    velocity;          // velocity of the vessel
8      std::vector<T>    acceleration;      // acceleration of vessel
9      std::vector<T>    pointing_direction; // AUV's pointing direction
10
11     // uniform linear-arrays
12     ULAClass<T>        ULA_fls;           // front-looking SONAR ULA
13     ULAClass<T>        ULA_portside;      // mounted ULA [object of class, ULAClass]
14     ULAClass<T>        ULA_starboard;     // mounted ULA [object of class, ULAClass]
15
16     // transmitters
17     TransmitterClass<T> transmitter_fls;   // transmitter for front-looking SONAR
18     TransmitterClass<T> transmitter_portside; // portside transmitter
19     TransmitterClass<T> transmitter_starboard; // starboard transmitter
20
21     // derived or dependent attributes
22     std::vector<std::vector<T>> signalMatrix_1; // matrix containing the
23         signals obtained from ULA_1
24     std::vector<std::vector<T>> largeSignalMatrix_1; // matrix holding signal of
25         synthetic aperture
26     std::vector<std::vector<T>> beamformedLargeSignalMatrix; // each column is the
27         beamformed signal at each stop-hop
28
29     // plotting mode
30     bool plottingmode; // to suppress plotting associated with classes
31
32     // spotlight mode related
33     std::vector<std::vector<T>> absolute_coords_patch_cart; // cartesian coordinates of

```



```

    patch
31
32 // Synthetic Aperture Related
33 std::vector<std::vector<T>> ApertureSensorLocations; // sensor locations of
    aperture
34
35 // functions
36 void syncComponentAttributes();
37 // void init(boost::asio::thread_pool& thread_pool);
38 void init(svr::ThreadPool& thread_pool);
39 // void simulate_signal(const ScattererClass<T>& seafloor,
40 //                       boost::asio::thread_pool& thread_pool);
41 void simulate_signal(const ScattererClass<T>& seafloor,
42                     svr::ThreadPool& thread_pool);
43 // void subset_scatterers(const ScattererClass<T>& seafloor,
44 //                       boost::asio::thread_pool& thread_pool);
45 void subset_scatterers(const ScattererClass<T>& seafloor,
46                       svr::ThreadPool& thread_pool,
47                       std::vector<std::size_t>& fls_scatterer_indices,
48                       std::vector<std::size_t>& portside_scatterer_indices,
49                       std::vector<std::size_t>& starboard_scatterer_indices);
50 void step(T time_step);
51
52 };

```

4.2 AUV Setup Scripts

The following script shows the setup-script for Uniform Linear Arrays

```

1 template <typename T>
2 void fAUVSetup(AUVClass<T>& auv) {
3
4     // building properties for the auv
5     auto location          {std::vector<T>{0, 50, 30}}; // starting location
6     auto velocity          {std::vector<T>{5, 0, 0}}; // starting velocity
7     auto pointing_direction {std::vector<T>{1, 0, 0}}; // pointing direction
8
9     // assigning
10    auv.location          = std::move(location); // assigning location
11    auv.velocity          = std::move(velocity); // assigning velocity
12    auv.pointing_direction = std::move(pointing_direction); // assigning pointing
        direction
13
14 }

```

Part II

Signal Simulation Pipeline

Part III

Imaging Pipeline

Part IV

Control Pipeline

Appendix A

General Purpose Templated Functions

A.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
11    // column
12    if constexpr(std::is_same_v<T, std::complex<double>> ||
13                 std::is_same_v<T, std::complex<float>> ||
14                 std::is_same_v<T, std::complex<long double>>){
15        for(int i = 0; i<inputvector.size(); ++i){
16            // adding entry
17            fileobj << inputvector[i].real() << "+" << inputvector[i].imag() << "i";
18
19            // adding delimiter
20            if(i!=inputvector.size()-1) {fileobj << ",";}
21            else {fileobj << "\n";}
22        }
23    }
24    else{
25        for(int i = 0; i<inputvector.size(); ++i){
26            fileobj << inputvector[i];
27            if(i!=inputvector.size()-1) {fileobj << ",";}
28            else {fileobj << "\n";}
29        }
30    }
31
32    // return
33    return;
34 }
35 // Matrix writing =====
36 template <typename T>
37 auto fWriteMatrix(const std::vector<std::vector<T>> inputMatrix,
38                  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 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() <<
70                     "i";
71                 if (j!=inputMatrix[0].size()-1) {fileobj << ",";}
72                 else {fileobj << "\n";}
73             }
74         }
75     }
76     else{
77         cout << format("File-write to {} failed\n", filename);
78     }
79 }

```

A.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(),

```



```

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 }

```

A.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,
6                 std::vector<T> >
7 {
8     // creating canvas vector
9     auto num_elements {input_vector_A.size() + input_vector_B.size()};
10    auto canvas {std::vector<T>(num_elements, (T)0) };

```



```

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     // moving it back
69     return std::move(canvas);
70
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
78                 == 0, 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 }

```

A.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 }

```

A.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 }

```

A.7 Coordinate Change

```

1 namespace svr {
2     /*=====
3     y = cart2sph(vector)
4     -----*/
5     template <typename T>
6     auto    cart2sph(const    std::vector<T>& cartesian_vector){
7
8         // splatting the point onto xy-plane
9         auto    xysplat    {cartesian_vector};
10        xysplat[2]    =    0;
11
12        // finding splat lengths
13        auto    xysplat_lengths    {norm(xysplat)};
14
15        // finding azimuthal and elevation angles
16        auto    azimuthal_angles    {svr::atan2(xysplat[1], xysplat[0]) *
17                                     180.00/std::numbers::pi};
18        auto    elevation_angles    {svr::atan2(cartesian_vector[2], xysplat_lengths) *
19                                     180.00/std::numbers::pi};
20        auto    rho_values          {norm(cartesian_vector)};
21
22        // creating tensor to send back
23        auto    spherical_vector    {std::vector<T>{azimuthal_angles,
24                                                     elevation_angles,
25                                                     rho_values}};
26
27        // moving it back
28        return std::move(spherical_vector);
29    }
30    /*=====
31    y = cart2sph(vector)
32    -----*/
33    template <typename T>
34    auto    cart2sph_inplace(std::vector<T>& cartesian_vector){
35
36        // splatting the point onto xy-plane
37        auto    xysplat    {cartesian_vector};
38
39

```

```

36     xysplat[2]      = 0;
37
38     // finding splat lengths
39     auto xysplat_lengths {norm(xysplat)};
40
41     // finding azimuthal and elevation angles
42     auto azimuthal_angles {svr::atan2(xysplat[1], xysplat[0]) *
43         180.00/std::numbers::pi};
44     auto elevation_angles {svr::atan2(cartesian_vector[2],
45         xysplat_lengths) * 180.00/std::numbers::pi};
46     auto rho_values      {norm(cartesian_vector)};
47
48     // creating tesnor
49     cartesian_vector[0]  = azimuthal_angles;
50     cartesian_vector[1]  = elevation_angles;
51     cartesian_vector[2]  = rho_values;
52 }
53 /*=====
54 y = cart2sph(input_matrix, dim)
55 -----*/
56 template <typename T>
57 auto cart2sph(const std::vector<std::vector<T>>& input_matrix,
58               const std::size_t axis)
59 {
60     // fetching dimensions
61     const auto& num_rows {input_matrix.size()};
62     const auto& num_cols {input_matrix[0].size()};
63
64     // checking the axis and dimensions
65     if (axis == 0 && num_rows != 3) {std::cerr << "cart2sph: incorrect num-elements
66         \n";}
67     if (axis == 1 && num_cols != 3) {std::cerr << "cart2sph: incorrect num-elements
68         \n";}
69
70     // creating canvas
71     auto canvas {std::vector<std::vector<T>>(
72         num_rows,
73         std::vector<T>(num_cols, 0)
74     )};
75
76     // if axis = 0, performing operation column-wise
77     if (axis == 0)
78     {
79         for(auto col = 0; col < num_cols; ++col)
80         {
81             // fetching current column
82             auto curr_column {std::vector<T>({input_matrix[0][col],
83                 input_matrix[1][col],
84                 input_matrix[2][col]})};
85
86             // performing inplace transformation
87             cart2sph_inplace(curr_column);
88
89             // storing it back
90             canvas[0][col] = curr_column[0];
91             canvas[1][col] = curr_column[1];
92             canvas[2][col] = curr_column[2];
93         }
94     }
95 }

```

```

92     // if axis == 1, performing operations row-wise
93     else if(axis == 0)
94     {
95         std::cerr << "cart2sph: yet to be implemented \n";
96     }
97     else
98     {
99         std::cerr << "cart2sph: yet to be implemented \n";
100     }
101
102     // returning
103     return std::move(canvas);
104
105 }
106
107 // =====
108 template <typename T>
109 auto sph2cart(const std::vector<T> spherical_vector){
110
111     // creating cartesian vector
112     auto cartesian_vector {std::vector<T>(spherical_vector.size(), 0)};
113
114     // populating
115     cartesian_vector[0] = spherical_vector[2] * \
116                          cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
117                          cos(spherical_vector[0] * std::numbers::pi / 180.00);
118     cartesian_vector[1] = spherical_vector[2] * \
119                          cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
120                          sin(spherical_vector[0] * std::numbers::pi / 180.00);
121     cartesian_vector[2] = spherical_vector[2] * \
122                          sin(spherical_vector[1] * std::numbers::pi / 180.00);
123
124     // returning
125     return std::move(cartesian_vector);
126 }
127 }

```

A.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 }

```

A.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 };

```

A.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 }

```

A.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 }

```

A.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 }

```

A.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     // throwing an error
31     if (nfft < input_vector.size()) {std::cerr << "size-mismatch\n";}
32     if (nfft <= 0) {std::cerr << "size-mismatch\n";}
33
34     // fetching data-type
35     using RType = fft_result_t<T>;
36     using baseType = std::conditional_t<std::is_same_v<T, std::complex<double>>,
37                                         double,
38                                         T>;
39
40     // canvas instantiation
41     std::vector<RType> canvas(nfft);
42     auto nfft_sqrt = {static_cast<RType>(std::sqrt(nfft))};
43     auto finaloutput = {std::vector<RType>(nfft, 0)};
44
45     // calculating index by index
46     for(int frequency_index = 0; frequency_index<nfft; ++frequency_index){
47         RType accumulate_value;
48         for(int signal_index = 0; signal_index < input_vector.size(); ++signal_index){
49             accumulate_value += \
50                 static_cast<RType>(input_vector[signal_index]) * \
51                 static_cast<RType>(std::exp(-1.00 * std::numbers::pi * \
52                                         (static_cast<baseType>(frequency_index)/static_cast<baseType>(nfft) * \
53                                         * \
54                                         static_cast<baseType>(signal_index))));
55         }
56         finaloutput[frequency_index] = accumulate_value / nfft_sqrt;
57     }
58
59     // returning
60     return std::move(finaloutput);
61 }

```



```

61
62 // =====
63 // y = ifft(x, nfft)
64 template<typename T>
65 auto ifft(const std::vector<T>& input_vector)
66 {
67     // fetching data-type
68     using RType = fft_result_t<T>;
69     using baseType = std::conditional_t<std::is_same_v<T, std::complex<double>>,
70                                         double,
71                                         T>;
72
73     // setup
74     auto nfft = {input_vector.size()};
75
76     // canvas instantiation
77     std::vector<RType> canvas(nfft);
78     auto nfft_sqrt = {static_cast<RType>(std::sqrt(nfft))};
79     auto finaloutput = {std::vector<RType>(nfft, 0)};
80
81     // calculating index by index
82     for(int frequency_index = 0; frequency_index<nfft; ++frequency_index){
83         RType accumulate_value;
84         for(int signal_index = 0; signal_index < input_vector.size(); ++signal_index){
85             accumulate_value += \
86                 static_cast<RType>(input_vector[signal_index]) * \
87                 static_cast<RType>(std::exp(1.00 * std::numbers::pi * \
88                                         (static_cast<baseType>(frequency_index)/static_cast<baseType>(nfft) * \
89                                         * \
90                                         static_cast<baseType>(signal_index))));
91         }
92         finaloutput[frequency_index] = accumulate_value / nfft_sqrt;
93     }
94
95     // returning
96     return std::move(finaloutput);
97 }

```

A.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 }

```

A.15 Indexing

```

1 namespace svr {
2     /*=====
3     y = index(vector, mask)
4     -----*/
5     template <typename T1,
6               typename T2,
7               typename = std::enable_if_t<std::is_arithmetic_v<T1>          ||
8                                           std::is_same_v<T1, std::complex<float> > ||
9                                           std::is_same_v<T1, std::complex<double> >
10                                           >
11                                           >
12     auto index(const std::vector<T1>& input_vector,
13               const std::vector<T2>& indices_to_sample)
14     {
15         // creating canvas
16         auto canvas {std::vector<T1>(indices_to_sample.size(), 0)};
17
18         // copying the associated values
19         for(int i = 0; i < indices_to_sample.size(); ++i){
20             auto source_index {indices_to_sample[i]};
21             if(source_index < input_vector.size()){
22                 canvas[i] = input_vector[source_index];
23             }
24             else
25                 cout << "svr::index | source_index !< input_vector.size()\n";
26         }
27
28         // returning
29         return std::move(canvas);
30     }
31     /*=====
32     y = index(matrix, mask, dim)
33     -----*/
34     template <typename T1, typename T2>
35     auto index(const std::vector<std::vector<T1>>& input_matrix,
36               const std::vector<T2>& indices_to_sample,
37               const std::size_t& dim)
38     {
39         // fetching dimensions
40         const auto& num_rows_matrix {input_matrix.size()};
41         const auto& num_cols_matrix {input_matrix[0].size()};
42
43         // creating canvas
44         auto canvas {std::vector<std::vector<T1>>>()};
45
46         // if indices are row-indices
47         if (dim == 0){
48
49             // initializing canvas
50             canvas = std::vector<std::vector<T1>>>(
51                 num_rows_matrix,
52                 std::vector<T1>(indices_to_sample.size())
53             );
54
55             // filling the canvas
56             auto destination_index {0};
57             std::for_each(indices_to_sample.begin(), indices_to_sample.end(),

```

```

58         [&](const auto& col){
59             for(auto row = 0; row < num_rows_matrix; ++row)
60                 canvas[row][destination_index] = input_matrix[row][col];
61             ++destination_index;
62         });
63     }
64     else if(dim == 1){
65         // initializing canvas
66         canvas = std::vector<std::vector<T1>>>(
67             indices_to_sample.size(),
68             std::vector<T1>(num_cols_matrix)
69         );
70
71         // filling the canvas
72         #pragma omp parallel for
73         for(auto row = 0; row < canvas.size(); ++row){
74             auto destination_col {0};
75             std::for_each(indices_to_sample.begin(), indices_to_sample.end(),
76                 [&row,
77                  &input_matrix,
78                  &destination_col,
79                  &canvas](const auto& source_col){
80                 canvas[row][destination_col++] =
81                     input_matrix[row][source_col];
82             });
83         }
84
85         // moving it back
86         return std::move(canvas);
87     }
88 }

```

A.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 +
34         static_cast<T>(i)*stepsize;}
35
36     return input;
37 };
38
39 // return-type
40 template <typename T, typename U>
41 auto linspace(T          startvalue,
42              U          endvalue,
43              size_t      numpoints)
44 {
45     vector<double> input(numpoints);
46     auto stepsize = static_cast<double>(endvalue -
47         startvalue)/static_cast<double>(numpoints-1);
48
49     for(int i = 0; i<input.size(); ++i) {input[i] = startvalue + i*stepsize;}
50
51     return input;
52 };

```

A.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 ==
3      1, std::vector<std::vector<T>> >
4  {
5      // setting up canvas
6      auto canvas
7          {std::vector<std::vector<T>>(input_matrix.size(),std::vector<T>(1))};
8
9      // filling up the canvas
10     for(auto row = 0; row < input_matrix.size(); ++row)
11         canvas[row][0] = *(std::max_element(input_matrix[row].begin(),
12             input_matrix[row].end()));
13
14     // returning
15     return std::move(canvas);
16 }

```

A.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 }

```

A.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,
   std::vector<std::vector<T>>> >
3 {
4     // creating canvas
5     auto canvas
6         {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(),
11         input_matrix[row].end()));

```

```

10
11     // returning the value
12     return std::move(canvas);
13 }

```

A.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  */

```

A.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) /
15                          static_cast<double>(input_scalar);
16                   });
17
18     // returning value
19     return std::move(canvas);
20 }
21 // =====
22 // matrix division with scalars
23 template <typename T>
24 auto operator/=(const std::vector<T>& input_vector,
25                const T& input_scalar)
26 {
27     // creating canvas
28     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) /
34                      static_cast<double>(input_scalar);
35               });
36
37 // returning value
38 return std::move(canvas);
39 }

```

A.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)};

```



```

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};
104
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 }

```

A.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-desparity

```

```

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
103 // scalar * matrix =====
104 template <typename T>
105 auto operator*(T scalar,
106               const std::vector<std::vector<T>>& inputMatrix)
107 {
108     std::vector<std::vector<T>> temp {inputMatrix};
109     for(int i = 0; i<inputMatrix.size(); ++i){
110         std::transform(inputMatrix[i].begin(),
111                       inputMatrix[i].end(),
112                       temp[i].begin(),
113                       [&scalar](T x){return scalar * x;});
114     }
115     return temp;

```

```

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 // scalar * matrix =====
155 template <typename T1, typename T2>
156 auto operator*(T1 scalar,
157               const std::vector<std::vector<T2>>& inputMatrix)
158 {
159     std::vector<std::vector<T2>> temp {inputMatrix};
160     for(int i = 0; i<inputMatrix.size(); ++i){
161         std::transform(inputMatrix[i].begin(),
162                       inputMatrix[i].end(),
163                       temp[i].begin(),
164                       [&scalar](T2 x){return static_cast<T2>(scalar) * x;});
165     }
166     return temp;
167 }
168 // matrix-multiplication =====
169 template <typename T1, typename T2>
170 auto matmul(const std::vector<std::vector<T1>>& matA,
171            const std::vector<std::vector<T2>>& matB)
172 {
173
174     // throwing error

```

```

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

```

```

230     std::cerr << "Operator*: [matrix, vector] | not implemented \n";
231 }
232
233 // returning
234 return std::move(canvas);
235
236 }
237
238 // scalar operators =====
239 auto operator*(const std::complex<double> complexscalar,
240               const double doublescalar){
241     return complexscalar * static_cast<std::complex<double>>(doublescalar);
242 }
243 auto operator*(const double doublescalar,
244               const std::complex<double> complexscalar){
245     return complexscalar * static_cast<std::complex<double>>(doublescalar);
246 }
247 auto operator*(const std::complex<double> complexscalar,
248               const int scalar){
249     return complexscalar * static_cast<std::complex<double>>(scalar);
250 }
251 auto operator*(const int scalar,
252               const std::complex<double> complexscalar){
253     return complexscalar * static_cast<std::complex<double>>(scalar);
254 }

```

A.24 Subtraction

```

1  /*=====
2  y = vector - scalar
3  -----*/
4  template <typename T>
5  auto operator-(const std::vector<T>& a, const T scalar){
6      std::vector<T> temp(a.size());
7      std::transform(a.begin(),
8                    a.end(),
9                    temp.begin(),
10                   [scalar](T x){return (x - scalar);});
11     return std::move(temp);
12 }
13 /*=====
14 y = vector - vector
15 -----*/
16 template <typename T>
17 auto operator-(const std::vector<T>& input_vector_A,
18               const std::vector<T>& input_vector_B)
19 {
20     // throwing error
21     if (input_vector_A.size() != input_vector_B.size())
22         std::cerr << "operator-(vector, vector): size disparity\n";
23
24     // creating canvas
25     const auto& num_cols {input_vector_A.size()};
26     auto canvas {std::vector<T>()};
27
28     // performing operations

```

```

29     std::transform(input_vector_A.begin(), input_vector_A.begin(),
30                    input_vector_B.begin(),
31                    canvas.begin(),
32                    [](const auto& argx, const auto& argy){
33                        return argx - argy;
34                    });
35
36     // return
37     return std::move(canvas);
38 }
39 /*=====
40 y = matrix - matrix
41 -----*/
42 template <typename T>
43 auto operator-(const std::vector<std::vector<T>>& input_matrix_A,
44               const std::vector<std::vector<T>>& input_matrix_B)
45 {
46     // fetching dimensions
47     const auto& num_rows_A {input_matrix_A.size()};
48     const auto& num_cols_A {input_matrix_A[0].size()};
49     const auto& num_rows_B {input_matrix_B.size()};
50     const auto& num_cols_B {input_matrix_B[0].size()};
51
52     // creating canvas
53     auto canvas {std::vector<std::vector<T>>()};
54
55     // if both matrices are of equal dimensions
56     if (num_rows_A == num_rows_B && num_cols_A == num_cols_B)
57     {
58         // copying one to the canvas
59         canvas = input_matrix_A;
60
61         // subtracting
62         for(auto row = 0; row < num_rows_B; ++row)
63             std::transform(canvas[row].begin(), canvas[row].end(),
64                            input_matrix_B[row].begin(),
65                            canvas[row].begin(),
66                            [](auto& argx, const auto& argy){
67                                return argx - argy;
68                            });
69     }
70     // column broadcasting (case 1)
71     else if(num_rows_A == num_rows_B && num_cols_B == 1)
72     {
73         // copying canvas
74         canvas = input_matrix_A;
75
76         // subtracting
77         for(auto row = 0; row < num_rows_A; ++row){
78             std::transform(canvas[row].begin(), canvas[row].end(),
79                            canvas[row].begin(),
80                            [&input_matrix_B,
81                             &row](auto& argx){
82                                return argx - input_matrix_B[row][0];
83                            });
84         }
85     }
86     else{
87         std::cerr << "operator-: not implemented for this case \n";

```

```

88     }
89
90     // returning
91     return std::move(canvas);
92 }

```

A.25 Operator Overloadings

A.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("[{}],{} | ", 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(),
75         inputMatrix[0].size());
76 }

```

A.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 final output
20     vector<T> finaloutput(numelements);

```

```

21     for(int i = 0; i<finaloutput.size(); ++i) {finaloutput[i] =
22         static_cast<T>(dist(gen));}
23
24     return finaloutput;
25 }
26 // =====
27 template <typename T>
28 auto rand(const T          argmin,
29           const T          argmax,
30           const vector<int> dimensions)
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 auto rand(const vector<int> dimensions)
55 {
56
57     using ReturnType = double;
58
59     // throwing an error if dimension is greater than two
60     if (dimensions.size() > 2) {std::cerr << "dimensions are too high\n";}
61
62     // creating random engine
63     static std::random_device rd; // Seed
64     static std::mt19937 gen(rd()); // Mersenne Twister generator
65     std::uniform_real_distribution<> dist(0.00, 1.00);
66
67     // building the finaloutput
68     vector<vector<ReturnType>> finaloutput;
69     for(int i = 0; i<dimensions[0]; ++i){
70         vector<ReturnType> temp;
71         for(int j = 0; j<dimensions[1]; ++j) {temp.push_back(dist(gen));}
72         finaloutput.push_back(std::move(temp));
73     }
74
75     // returning the finaloutput
76     return std::move(finaloutput);
77 }
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)
100             {temp.push_back(static_cast<double>(dist(gen)));}
101         finaloutput.push_back(std::move(temp));
102     }
103
104     // returning the finaloutput
105     return finaloutput;
106 }

```

A.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 }

```

A.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,
4         std::vector<T>>
5 {
6     // returning the input as is
7     return input_vector;
8 }
9 // =====
10 template <std::size_t axis, typename T>
11 auto sum(const std::vector<std::vector<T>>& input_matrix) -> std::enable_if_t<axis == 0,
12         std::vector<T>>
13 {
14     // creating canvas
15     auto canvas {std::vector<T>(input_matrix[0].size(), 0)};
16
17     // filling up the canvas
18     for(auto row = 0; row < input_matrix.size(); ++row)
19         std::transform(input_matrix[row].begin(), input_matrix[row].end(),
20             canvas.begin(),
21             canvas.begin(),
22             [](auto& argx, auto& argy){return argx + argy;});
23
24     // returning
25     return std::move(canvas);
26 }
27 // =====
28 template <std::size_t axis, typename T>

```

```

28 auto sum(const std::vector<std::vector<T>>& input_matrix) -> std::enable_if_t<axis == 1,
    std::vector<std::vector<T>>>
29 {
30     // creating canvas
31     auto canvas {std::vector<std::vector<T>>(input_matrix.size(),
32                                             std::vector<T>(1, 0.00))};
33
34     // filling up the canvas
35     for(auto row = 0; row < input_matrix.size(); ++row)
36         canvas[row][0] = std::accumulate(input_matrix[row].begin(),
37                                           input_matrix[row].end(),
38                                           static_cast<T>(0));
39
40     // returning
41     return std::move(canvas);
42
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 }

```

A.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 }

```

A.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  }

```

A.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 }

```

A.33 Masking

```

1  namespace svr {
2      /*=====
3      y = input_vector[mask == 1]

```



```

4  -----*/
5  template <typename T,
6          typename = std::enable_if_t< std::is_arithmetic_v<T>          ||
7          std::is_same_v<T, std::complex<double>> ||
8          std::is_same_v<T, std::complex<float>>
9          >
10         >
11  auto mask(const std::vector<T>& input_vector,
12           const std::vector<bool>& mask_vector)
13  {
14      // checking dimensionality issues
15      if (input_vector.size() != mask_vector.size())
16          std::cerr << "mask(vector, mask): incompatible size \n";
17
18      // creating canvas
19      auto num_trues {std::count(mask_vector.begin(),
20                               mask_vector.end(),
21                               true)};
22      auto canvas {std::vector<T>(num_trues)};
23
24      // copying values
25      auto destination_index {0};
26      for(auto i = 0; i < input_vector.size(); ++i)
27          if (mask_vector[i] == true)
28              canvas[destination_index++] = input_vector[i];
29
30      // returning output
31      return std::move(canvas);
32  }
33  /*=====
34  -----*/
35  template <typename T>
36  auto mask(const std::vector<std::vector<T>>& input_matrix,
37           const std::vector<bool> mask_vector)
38  {
39      // fetching dimensions
40      const auto& num_rows_matrix {input_matrix.size()};
41      const auto& num_cols_matrix {input_matrix[0].size()};
42      const auto& num_cols_vector {mask_vector.size()};
43
44      // error-checking
45      if (num_cols_matrix != num_cols_vector)
46          std::cerr << "mask(matrix, bool-vector): size disparity";
47
48      // building canvas
49      auto num_trues {std::count(mask_vector.begin(),
50                               mask_vector.end(),
51                               true)};
52      auto canvas {std::vector<std::vector<T>>(
53          num_rows_matrix,
54          std::vector<T>(num_cols_vector, 0)
55      )};
56
57      // writing values
58      #pragma omp parallel for
59      for(auto row = 0; row < num_rows_matrix; ++row){
60          auto destination_index {0};
61          for(auto col = 0; col < num_cols_vector; ++col)
62              if(mask_vector[col] == true)

```

```

63         canvas[row][destination_index++] = input_matrix[row][col];
64     }
65
66     // returning
67     return std::move(canvas);
68 }
69 /*=====
70 Fetch Indices corresponding to mask true's
71 -----*/
72 auto mask_indices(const std::vector<bool>& mask_vector)
73 {
74     // creating canvas
75     auto num_trues {std::count(mask_vector.begin(), mask_vector.end(),
76                               true)};
77     auto canvas {std::vector<std::size_t>(num_trues)};
78
79     // building canvas
80     auto destination_index {0};
81     for(auto i = 0; i < mask_vector.size(); ++i)
82         if (mask_vector[i] == true)
83             canvas[destination_index++] = i;
84
85     // returning
86     return std::move(canvas);
87 }
88 }

```

A.34 Resetting Containers

```

1 namespace svr {
2     /*=====
3     -----*/
4     // template <typename T>
5     // void reset(std::vector<T>& input_vector) {
6     //     std::vector<T>().swap(input_vector);
7     // }
8     /*=====
9     Variadic version of resetting
10    -----*/
11    template <typename T, typename... Rest>
12    void reset(std::vector<T>& first_vector, Rest&... rest_vectors) {
13        // Reset the first vector
14        std::vector<T>().swap(first_vector);
15
16        // Recursively reset the remaining vectors
17        if constexpr (sizeof...(rest_vectors) > 0) {
18            reset(rest_vectors...);
19        }
20    }
21 }

```

A.35 Element-wise squaring

```

1 namespace svr {

```

```

2  /*=====
3  Element-wise squaring vector
4  -----*/
5  template <typename T,
6           typename = std::enable_if_t<std::is_arithmetic_v<T>>
7           >
8  auto square(const std::vector<T>& input_vector)
9  {
10     // creating canvas
11     auto canvas {std::vector<T>(input_vector.size())};
12
13     // performing calculations
14     std::transform(input_vector.begin(), input_vector.end(),
15                   canvas.begin(),
16                   [](const auto& argx){
17                       return argx * argx;
18                   });
19
20     // moving it back
21     return std::move(canvas);
22 }
23 /*=====
24 Element-wise squaring vector (in-place)
25 -----*/
26 template <typename T,
27           typename = std::enable_if_t<std::is_arithmetic_v<T>>
28           >
29 void square_inplace(std::vector<T>& input_vector)
30 {
31     // performing operations
32     std::transform(input_vector.begin(), input_vector.end(),
33                   input_vector.begin(),
34                   [](auto& argx){
35                       return argx * argx;
36                   });
37 }
38 /*=====
39 Element-wise squaring a matrix
40 -----*/
41 template <typename T>
42 auto square(const std::vector<std::vector<T>>& input_matrix)
43 {
44     // fetching dimensions
45     const auto& num_rows {input_matrix.size()};
46     const auto& num_cols {input_matrix[0].size()};
47
48     // creating canvas
49     auto canvas {std::vector<std::vector<T>>(
50         num_rows,
51         std::vector<T>(num_cols, 0)
52     )};
53
54     // going through each row
55     #pragma omp parallel for
56     for(auto row = 0; row < num_rows; ++row)
57         std::transform(input_matrix[row].begin(), input_matrix[row].end(),
58                       canvas[row].begin(),
59                       [](const auto& argx){
60                           return argx * argx;

```

```

61         });
62
63         // returning
64         return std::move(canvas);
65     }
66     /*=====
67     Squaring for scalars
68     -----*/
69     template <typename T>
70     auto square(const T& scalar)    {return scalar * scalar;}
71 }

```

A.36 Thread-Pool

```

1  namespace svr {
2      class ThreadPool {
3      public:
4          // Members
5          boost::asio::thread_pool    thread_pool;        // the pool
6          std::vector<std::future<void>> future_vector;    // futures to wait on
7
8          // Special-Members
9          ThreadPool(std::size_t num_threads)
10             : thread_pool(num_threads) {}
11          ThreadPool(const ThreadPool& other)    = delete;
12          ThreadPool& operator=(ThreadPool& other) = delete;
13
14          // Member-functions
15          void                converge();
16          template <typename F> void push_back(F&& func);
17          void                shutdown();
18
19      private:
20          template<typename F>
21          std::future<void> _wrap_task(F&& func) {
22              std::promise<void> p;
23              auto f = p.get_future();
24
25              boost::asio::post(thread_pool,
26                  [func = std::forward<F>(func), p = std::move(p)]() mutable {
27                      func();
28                      p.set_value();
29                  });
30
31              return f;
32          }
33      };
34
35      /*=====
36      Member-Function: Add new task to the pool
37      -----*/
38      template <typename F>
39      void ThreadPool::push_back(F&& func)
40      {
41          future_vector.push_back(_wrap_task(std::forward<F>(func)));
42      }

```

```
43  /*=====
44  Member-Function: waiting until all the assigned work is done
45  -----*/
46  void ThreadPool::converge()
47  {
48      for (auto &fut : future_vector) fut.get();
49      future_vector.clear();
50  }
51  /*=====
52  Member-Function: Shutting things down
53  -----*/
54  void ThreadPool::shutdown()
55  {
56      thread_pool.join();
57  }
58
59 }
```
