

Autonomous Underwater Vehicle: A Surveillance Protocol

S.V. Rajendran

October 7, 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 Perception & Control Pipeline	23
A General Purpose Templated Functions	24
A.1 abs	24

A.2	Boolean Comparators	25
A.3	Concatenate Functions	26
A.4	Conjugate	28
A.5	Convolution	28
A.6	Coordinate Change	34
A.7	Cosine	37
A.8	Data Structures	37
A.9	Editing Index Values	38
A.10	Equality	39
A.11	Exponentiate	39
A.12	FFT	40
A.13	Flipping Containers	45
A.14	Indexing	45
A.15	Linspace	47
A.16	Max	48
A.17	Meshgrid	48
A.18	Minimum	49
A.19	Norm	50
A.20	Division	52
A.21	Addition	53
A.22	Multiplication (Element-wise)	56
A.23	Subtraction	61
A.24	Operator Overloadings	63
A.25	Printing Containers	63
A.26	Random Number Generation	65
A.27	Reshape	67
A.28	Summing with containers	68
A.29	Tangent	70
A.30	Tiling Operations	71
A.31	Transpose	72
A.32	Masking	72
A.33	Resetting Containers	74
A.34	Element-wise squaring	74
A.35	Flooring	75
A.36	Squeeze	77

A.37	Tensor Initializations	78
A.38	Real part	78
A.39	Imaginary part	79
B	Software Tools	81
B.1	CSV File-Writes	81
B.2	Thread-Pool	83
B.3	FFTPlanClass	85
B.4	IFFTPlanClass	91
B.5	FFT Plan Pool	96
B.6	IFFT Plan Pool	98

Part I

AUV Components & Setup

Chapter 1

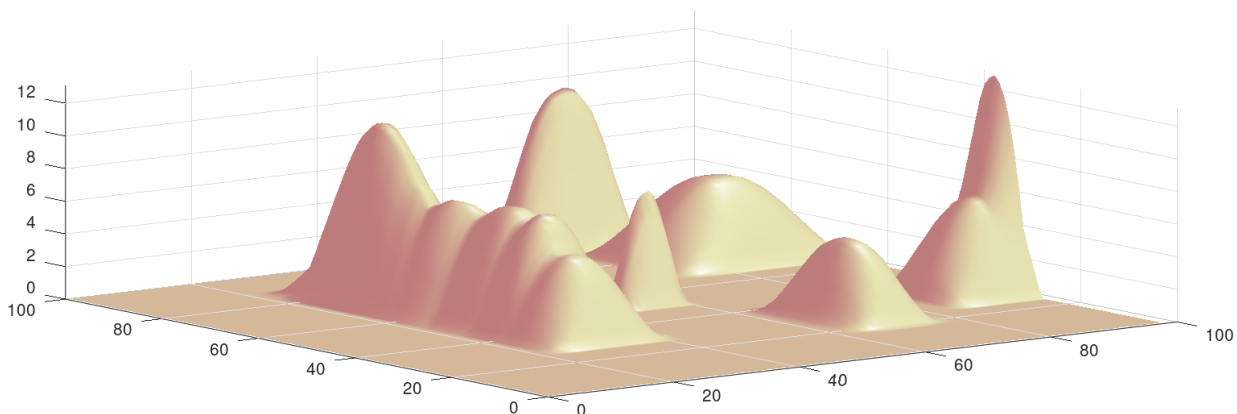
Underwater Environment

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    auto   bed_width_density {static_cast<double>( 5.00)};
19    auto   bed_length_density {static_cast<double>( 5.00)};
20
21    // setting up coordinates
22    auto   xpoints      {linspace<double>(0.00,
23                                         bed_width,
24                                         bed_width * bed_width_density)};
25    auto   ypoints      {linspace<double>(0.00,
26                                         bed_length,
27                                         bed_length * bed_length_density)};
28    if(save_files) fWriteVector(xpoints, "../csv-files/xpoints.csv"); // verified
29    if(save_files) fWriteVector(ypoints, "../csv-files/ypoints.csv"); // verified
30
31    // creating mesh
32    auto [xgrid, ygrid] = meshgrid(std::move(xpoints), std::move(ypoints));
33    if(save_files) fWriteMatrix(xgrid,  "../csv-files/xgrid.csv"); // verified
34    if(save_files) fWriteMatrix(ygrid,  "../csv-files/ygrid.csv"); // verified
35
36    // reshaping
37    auto   X      {reshape(xgrid, xgrid.size()*xgrid[0].size())};
38    auto   Y      {reshape(ygrid, ygrid.size()*ygrid[0].size())};
39    if(save_files) fWriteVector(X,      "../csv-files/X.csv"); // verified
40    if(save_files) fWriteVector(Y,      "../csv-files/Y.csv"); // verified
41
42    // creating heights of scatterers
43    if(hill_creation_flag){
44
45        // setting up hill parameters
46        auto   num_hills      {10};
47
48        // setting up placement of hills
49        auto   points2D      {concatenate<0>(X, Y)}; // verified

```

```

50     auto    min2D                {min<1, double>(points2D)};           // verified
51     auto    max2D                {max<1, double>(points2D)};           // verified
52     auto    hill_2D_center       {min2D + \
53                                   rand({2, num_hills}) * (max2D - min2D)}; // verified
54
55     // setup: hill-dimensions
56     auto    hill_dimensions_min   {transpose(vector<double>{5, 5, 2})}; // verified
57     auto    hill_dimensions_max   {transpose(vector<double>{30, 30, 10})}; // verified
58     auto    hill_dimensions       {hill_dimensions_min + \
59                                   rand({3, num_hills}) * (hill_dimensions_max -
60                                   hill_dimensions_min)};           // verified
61
62     // function-call: hill-creation function
63     fCreateHills(hill_2D_center,
64                  hill_dimensions,
65                  points2D);
66
67     // setting up floor reflectivity
68     auto    floorScatter_reflectivity {std::vector<double>(Y.size(), 1.00)};
69
70     // populating the values of the incoming argument
71     scatterers.coordinates = std::move(points2D);
72     scatterers.reflectivity = std::move(floorScatter_reflectivity);
73
74 }
75
76 else{
77
78     // assigning flat heights
79     auto    Z {std::vector<double>(Y.size(), 0)};
80
81     // setting up floor coordinates
82     auto    floorScatter_coordinates {concatenate<0>(X, Y, Z)};
83     auto    floorScatter_reflectivity {std::vector<double>(Y.size(), 1)};
84
85     // populating the values of the incoming argument
86     scatterers.coordinates = std::move(floorScatter_coordinates);
87     scatterers.reflectivity = std::move(floorScatter_reflectivity);
88
89 }
90
91 // printing status
92 std::cout << format("> Finished Sea-Floor Setup \n");
93 }

```

Chapter 2

Transmitter

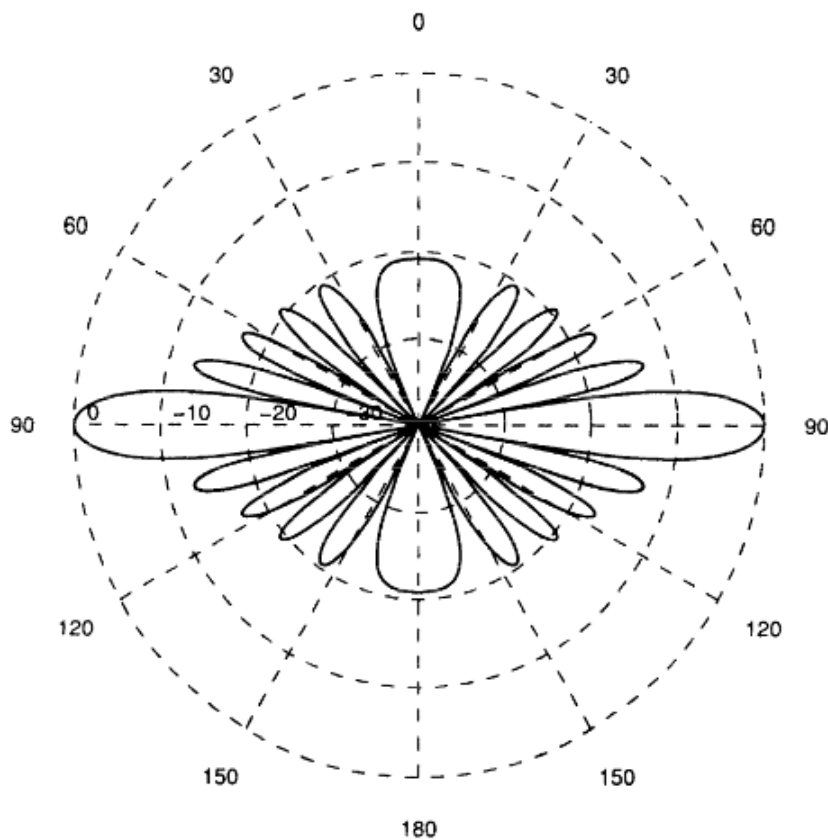


Figure 2.1: Beampattern of a Transmission Uniform Linear Array

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

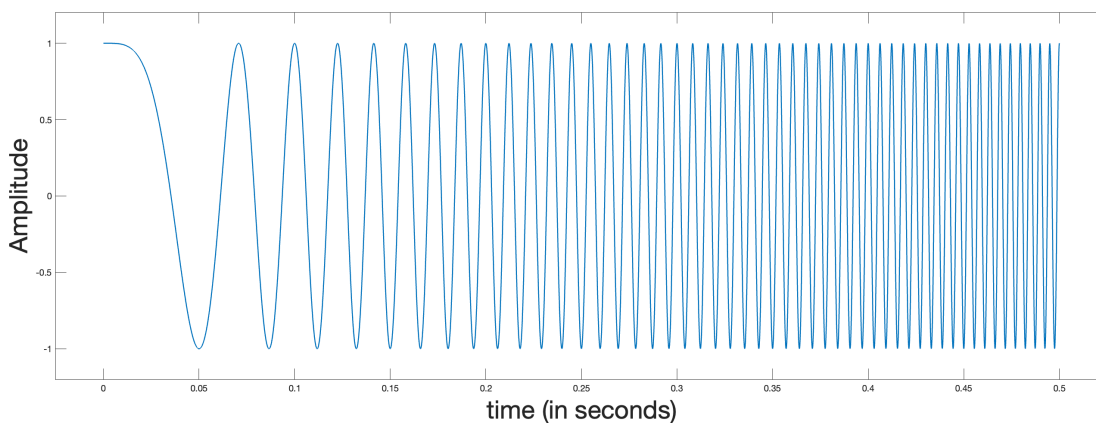


Figure 2.2: Linear Frequency Modulated Wave

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      // A shared pointer to the configuration object
6      std::shared_ptr<svr::AUVParameters> config_ptr;
7
8      // physical/intrinsic properties
9      std::vector<T>    location;           // location tensor
10     std::vector<T>    pointing_direction; // pointing direction
11
12     // basic parameters
13     std::vector<T>    signal;             // transmitted signal (LFM)
14     T                 azimuthal_angle;    // transmitter's azimuthal pointing direction
15     T                 elevation_angle;    // transmitter's elevation pointing direction
16     T                 azimuthal_beamwidth; // azimuthal beamwidth of transmitter
17     T                 elevation_beamwidth; // elevation beamwidth of transmitter
18     T                 range;             // a parameter used for spotlight mode.
19
20     // transmitted signal attributes
21     T                 f_low;             // lowest frequency of LFM
22     T                 f_high;           // highest frequency of LFM
23     T                 fc;               // center frequency of LFM
24     T                 bandwidth;        // bandwidth of LFM
25     T                 speed_of_sound {1500}; // speed of sound
26
27     // shadowing properties
28     int               azimuthQuantDensity; // quantization of angles along the
29     azimuth           elevationQuantDensity; // quantization of angles along the
30     elevation
31     T                 rangeQuantSize;    // range-cell size when shadowing
32     T                 azimuthShadowThreshold; // azimuth thresholding
33     T                 elevationShadowThreshold; // elevation thresholding

```

```

33
34 // shadowing related
35 std::vector<T> checkbox; // box indicating whether a scatter for a
    range-angle pair has been found
36 std::vector<std::vector<std::vector<T>>> finalScatterBox; // a 3D tensor where the
    third dimension represnets the vector length
37 std::vector<T> finalReflectivityBox; // to store the reflectivity
38
39 // constructor
40 TransmitterClass() = default;
41
42 // Deleting copy constructors/assignment
43 TransmitterClass(const TransmitterClass& other) = delete;
44 TransmitterClass& operator=(TransmitterClass& other) = delete;
45
46 // Creating move-constructor and move-assignment
47 TransmitterClass(TransmitterClass&& other) = default;
48 TransmitterClass& operator=(TransmitterClass&& other) = default;
49
50 // member-functions
51 auto updatePointingAngle(std::vector<T> AUV_pointing_vector);
52 auto subset_scatterers(const ScattererClass<T>& seafloor,

```

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     // signaling status
107     std::cout << format("> Finished Transmitter Setups\n");
108
109 }
```

Chapter 3

Uniform Linear Array

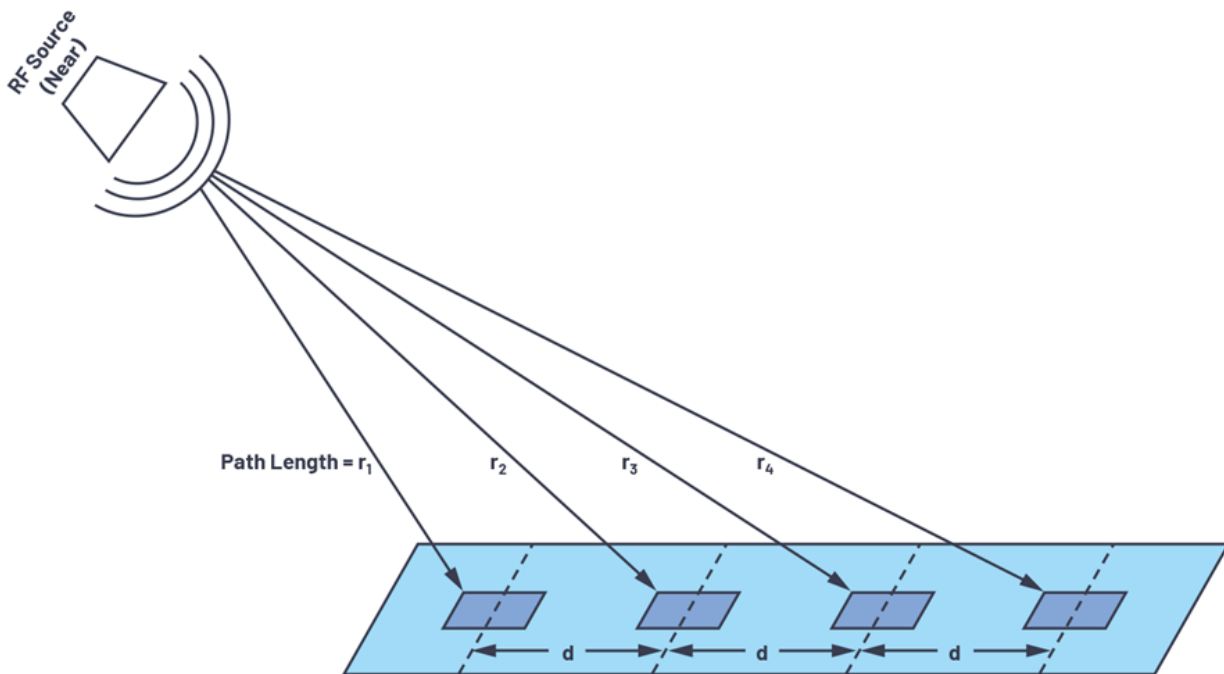


Figure 3.1: Uniform Linear Array

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      std::size_t                num_sensors;                // number
7      T                          inter_element_spacing;      // space between
8      std::vector<std::vector<T>> coordinates;                // coordinates
9      T                          sampling_frequency;          // sampling
10     T                          recording_period;             // recording
11     std::vector<T>              location;                     // location of
12     // derived
13     std::vector<T>              sensor_direction;
14     std::vector<std::vector<T>> signal_matrix;
15
16     // decimation related
17     int                          decimation_factor;          // the new decimation
18     T                            post_decimation_sampling_frequency; // the new sampling
19     std::vector<T>              lowpass_filter_coefficients_for_decimation; // filter-coefficients
20     // imaging related
21     T range_resolution;      // theoretical range-resolution =  $\frac{c}{2B}$ 
22     T azimuthal_resolution;  // theoretical azimuth-resolution =
23     T range_cell_size;       // the range-cell quanta we're choosing for
24     T azimuth_cell_size;     // the azimuth quanta we're choosing
25     std::vector<T> azimuth_centers; // tensor containing the azimuth centers
26     std::vector<T> range_centers;  // tensor containing the range-centers
27     int frame_size;           // the frame-size corresponding to a range cell in a
28     decimated signal matrix

```

```

31  std::vector<std::vector<complex<T>>> mulFFTMatrix; // 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      signal_matrix(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::norm(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  // // deleting copy constructor/assignment
80  // ULAClass<T>(const ULAClass<T>& other) = delete;
81  // ULAClass<T>& operator=(const ULAClass<T>& other) = delete;
82  ULAClass<T>(ULAClass<T>&& other) = delete;
83  ULAClass<T>& operator=(const ULAClass<T>& other) = default;

```

```

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

```

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>(16)};          // 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     assigning number of sensors
42     ula_fls.inter_element_spacing = inter_element_spacing;    //
43     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
56     // signaling end
57     std::cout << format("Finished ULA-Setup\n");
58 }

```

Chapter 4

Autonomous Underwater Vehicle

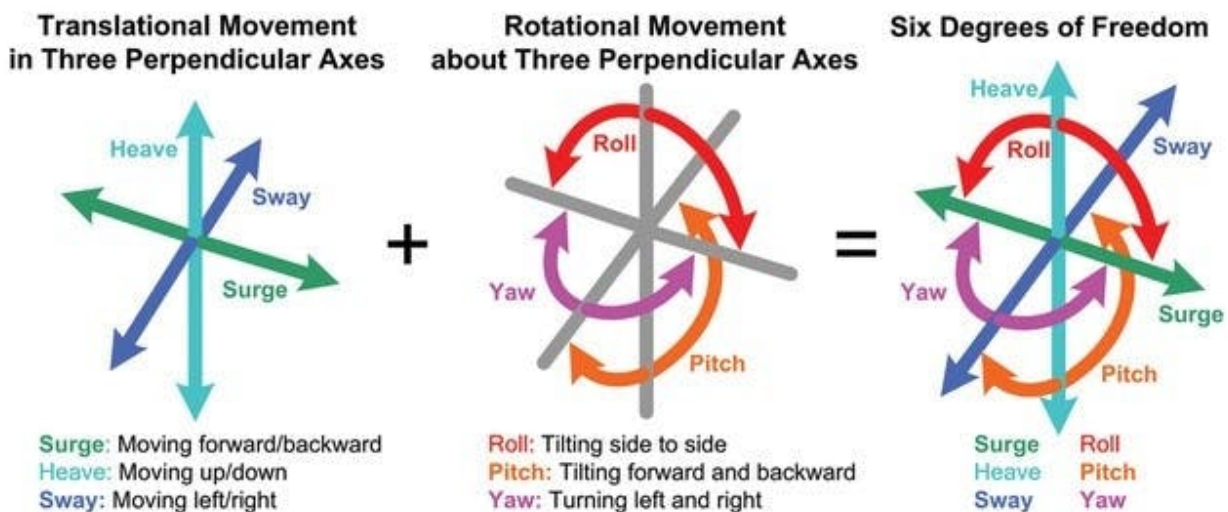


Figure 4.1: AUV degrees of freedom

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
        signals obtained from ULA_1
23     std::vector<std::vector<T>> largeSignalMatrix_1; // matrix holding signal of
        synthetic aperture
24     std::vector<std::vector<T>> beamformedLargeSignalMatrix; // each column is the
        beamformed signal at each stop-hop
25
26     // plotting mode
27     bool plottingmode; // to suppress plotting associated with classes
28
29     // spotlight mode related
30     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(svr::ThreadPool& thread_pool);
38 void simulate_signal(const ScattererClass<T>& seafloor,
39                     svr::ThreadPool& thread_pool);
40 void subset_scatterers(const ScattererClass<T>& seafloor,
41                       svr::ThreadPool& thread_pool,
42                       std::vector<std::size_t>& fls_scatterer_indices,
43                       std::vector<std::size_t>& portside_scatterer_indices,
44                       std::vector<std::size_t>& starboard_scatterer_indices);
45 void step(T time_step);
46
47 };
48
49 /*=====
50 Aim: update attributes
51 -----*/
52 template <typename T>

```

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    // signaling end
15    std::cout << format("> Completed AUV-setup\n");
16
17 }

```

Part II

Signal Simulation Pipeline

Part III

Imaging Pipeline

Part IV

Perception & Control Pipeline

Appendix A

General Purpose Templated Functions

A.1 abs

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

```

39         [](auto& argx){return std::abs(argx);});
40
41     // returning
42     return std::move(canvas);
43 }

```

A.2 Boolean Comparators

```

1  #pragma once
2  /*=====
3  -----*/
4  template <typename T, typename U>
5  auto operator<(const std::vector<T>& input_vector,
6                const U scalar)
7  {
8      // creating canvas
9      auto canvas {std::vector<bool>(input_vector.size())};
10
11     // transforming
12     std::transform(input_vector.begin(), input_vector.end(),
13                   canvas.begin(),
14                   [&scalar](const auto& argx){
15                       return argx < static_cast<T>(scalar);
16                   });
17
18     // returning
19     return std::move(canvas);
20 }
21 /*=====
22 -----*/
23 template <typename T, typename U>
24 auto operator<=(const std::vector<T>& input_vector,
25                const U scalar)
26 {
27     // creating canvas
28     auto canvas {std::vector<bool>(input_vector.size())};
29
30     // transforming
31     std::transform(input_vector.begin(), input_vector.end(),
32                   canvas.begin(),
33                   [&scalar](const auto& argx){
34                       return argx <= static_cast<T>(scalar);
35                   });
36
37     // returning
38     return std::move(canvas);
39 }
40 // =====
41 template <typename T, typename U>
42 auto operator>(const std::vector<T>& input_vector,
43                const U scalar)
44 {
45     // creating canvas
46     auto canvas {std::vector<bool>(input_vector.size())};
47
48     // transforming

```

```

49     std::transform(input_vector.begin(), input_vector.end(),
50                   canvas.begin(),
51                   [&scalar](const auto& argx){
52                       return argx > static_cast<T>(scalar);
53                   });
54
55     // returning
56     return std::move(canvas);
57 }
58 /*=====
59 -----*/
60 template <typename T, typename U>
61 auto operator>=(const std::vector<T>& input_vector,
62               const U scalar)
63 {
64     // creating canvas
65     auto canvas {std::vector<bool>(input_vector.size())};
66
67     // transforming
68     std::transform(input_vector.begin(), input_vector.end(),
69                   canvas.begin(),
70                   [&scalar](const auto& argx){
71                       return argx >= static_cast<T>(scalar);
72                   });
73
74     // returning
75     return std::move(canvas);
76 }

```

A.3 Concatenate Functions

```

1  #pragma once
2  /*=====
3  input = [vector, vector],
4  output = [vector]
5  -----*/
6  template <std::size_t axis, typename T>
7  auto concatenate(const std::vector<T>& input_vector_A,
8                 const std::vector<T>& input_vector_B) -> std::enable_if_t<axis == 1,
9                 std::vector<T> >
10 {
11     // creating canvas vector
12     auto num_elements {input_vector_A.size() + input_vector_B.size()};
13     auto canvas {std::vector<T>(num_elements, (T)0) };
14
15     // filling up the canvas
16     std::copy(input_vector_A.begin(), input_vector_A.end(),
17               canvas.begin());
18     std::copy(input_vector_B.begin(), input_vector_B.end(),
19               canvas.begin()+input_vector_A.size());
20
21     // moving it back
22     return std::move(canvas);
23 }
24 /*=====

```

```

25 input = [vector, vector],
26 output = [matrix]
27 -----*/
28 template <std::size_t axis, typename T>
29 auto concatenate(const std::vector<T>& input_vector_A,
30                 const std::vector<T>& input_vector_B) -> std::enable_if_t<axis == 0,
31                 std::vector<std::vector<T>>> >
32 {
33     // throwing error dimensions
34     if (input_vector_A.size() != input_vector_B.size())
35         std::cerr << "concatenate:: incorrect dimensions \n";
36
37     // creating canvas
38     auto canvas {std::vector<std::vector<T>>>(
39         2, std::vector<T>(input_vector_A.size())
40     )};
41
42     // filling up the dimensions
43     std::copy(input_vector_A.begin(), input_vector_A.end(), canvas[0].begin());
44     std::copy(input_vector_B.begin(), input_vector_B.end(), canvas[1].begin());
45
46     // moving it back
47     return std::move(canvas);
48 }
49 /*=====
50 input = [vector, vector, vector],
51 output = [matrix]
52 -----*/
53 template <std::size_t axis, typename T>
54 auto concatenate(const std::vector<T>& input_vector_A,
55                 const std::vector<T>& input_vector_B,
56                 const std::vector<T>& input_vector_C) -> std::enable_if_t<axis == 0,
57                 std::vector<std::vector<T>>> >
58 {
59     // throwing error dimensions
60     if (input_vector_A.size() != input_vector_B.size() ||
61         input_vector_A.size() != input_vector_C.size())
62         std::cerr << "concatenate:: incorrect dimensions \n";
63
64     // creating canvas
65     auto canvas {std::vector<std::vector<T>>>(
66         3, std::vector<T>(input_vector_A.size())
67     )};
68
69     // filling up the dimensions
70     std::copy(input_vector_A.begin(), input_vector_A.end(), canvas[0].begin());
71     std::copy(input_vector_B.begin(), input_vector_B.end(), canvas[1].begin());
72     std::copy(input_vector_C.begin(), input_vector_C.end(), canvas[2].begin());
73
74     // moving it back
75     return std::move(canvas);
76 }
77 /*=====
78 input = [matrix, vector],
79 output = [matrix]
80 -----*/
81 template <std::size_t axis, typename T>

```



```

82 auto concatenate(const std::vector<std::vector<T>>& input_matrix,
83                 const std::vector<T>          input_vector) -> std::enable_if_t<axis
                        == 0, std::vector<std::vector<T>>> >
84 {
85     // creating canvas
86     auto canvas {input_matrix};
87
88     // adding to the canvas
89     canvas.push_back(input_vector);
90
91     // returning
92     return std::move(canvas);
93 }

```

A.4 Conjugate

```

1  #pragma once
2  namespace svr {
3      /*=====
4      y = svr::conj(vector);
5      -----*/
6      template <typename T>
7      auto conj(const std::vector<T>& input_vector)
8      {
9          // creating canvas
10         auto canvas {std::vector<T>(input_vector.size())};
11
12         // calculating conjugates
13         std::for_each(canvas.begin(), canvas.end(),
14             [](auto& argx){argx = std::conj(argx);});
15
16         // returning
17         return std::move(canvas);
18     }
19 }

```

A.5 Convolution

```

1  #pragma once
2  namespace svr {
3
4      /*=====
5      1D convolution of two vectors
6      > implemented through fft
7      -----*/
8      template <typename T1, typename T2>
9      auto conv1D(const std::vector<T1>& input_vector_A,
10                 const std::vector<T2>& input_vector_B)
11      {
12          // resulting type
13          using T3 = decltype(std::declval<T1>() * std::declval<T2>());
14
15          // creating canvas
16          auto canvas_length {input_vector_A.size() + input_vector_B.size() - 1};

```

```

17
18 // calculating fft of two arrays
19 auto fft_A {svr::fft(input_vector_A, canvas_length)};
20 auto fft_B {svr::fft(input_vector_B, canvas_length)};
21
22 // element-wise multiplying the two matrices
23 auto fft_AB {fft_A * fft_B};
24
25 // finding inverse FFT
26 auto convolved_result {ifft(fft_AB)};
27
28 // returning
29 return std::move(convolved_result);
30 }
31
32 template <>
33 auto conv1D(const std::vector<double>& input_vector_A,
34            const std::vector<double>& input_vector_B)
35 {
36 // creating canvas
37 auto canvas_length {input_vector_A.size() + input_vector_B.size() - 1};
38
39 // calculating fft of two arrays
40 auto fft_A {svr::fft(input_vector_A, canvas_length)};
41 auto fft_B {svr::fft(input_vector_B, canvas_length)};
42
43 // element-wise multiplying the two matrices
44 auto fft_AB {fft_A * fft_B};
45
46 // finding inverse FFT
47 auto convolved_result {ifft(fft_AB)};
48
49 // returning
50 return std::move(convolved_result);
51 }
52
53 /*=====
54 1D convolution of two vectors
55 > implemented through fft
56 -----*/
57 template <typename T1, typename T2>
58 auto conv1D_fftw(const std::vector<T1>& input_vector_A,
59                 const std::vector<T2>& input_vector_B)
60 {
61 // resulting type
62 using T3 = decltype(std::declval<T1>() * std::declval<T2>());
63
64 // creating canvas
65 auto canvas_length {input_vector_A.size() + input_vector_B.size() - 1};
66
67 // calculating fft of two arrays
68 auto fft_A {svr::fft(input_vector_A, canvas_length)};
69 auto fft_B {svr::fft(input_vector_B, canvas_length)};
70
71 // element-wise multiplying the two matrices
72 auto fft_AB {fft_A * fft_B};
73
74 // finding inverse FFT
75 auto convolved_result {svr::ifft(fft_AB, fft_AB.size())};

```

```

76
77     // returning
78     return std::move(convolved_result);
79 }
80
81 /*=====
82 Long-signal Conv1D
83 improvements:
84 > make an inplace version of this
85 -----*/
86 template <std::size_t L, typename T>
87 auto conv1D_long(const std::vector<T>& input_vector_A,
88                 const std::vector<T>& input_vector_B)
89 {
90     // fetching dimensions
91     const auto maxlength = {std::max(input_vector_A.size(),
92                                     input_vector_B.size())};
93     const auto filter_size = {std::min(input_vector_A.size(),
94                                     input_vector_B.size())};
95     const auto block_size = {L + filter_size - 1};
96     const auto num_blocks = {2 + static_cast<std::size_t>(
97         (maxlength - block_size)/L
98     )};
99
100     // obtaining references
101     const auto& large_vector = {input_vector_A.size() >= input_vector_B.size() ? \
102                                input_vector_A      : input_vector_B};
103     const auto& small_vector = {input_vector_A.size() < input_vector_B.size() ? \
104                                input_vector_A      : input_vector_B};
105
106     // setup
107     auto starting_index = {static_cast<std::size_t>(0)};
108     auto ending_index = {static_cast<std::size_t>(0)};
109     auto length_left_to_fill = {ending_index - starting_index};
110     auto canvas = {std::vector<double>(block_size, 0)};
111     auto finaloutput = {std::vector<double>(maxlength, 0)};
112     auto block_conv_output_size = {L + 2 * filter_size - 2};
113     auto block_conv_output = {std::vector<double>(block_conv_output_size, 0)};
114
115     // block-wise processing
116     for(auto bid = 0; bid < num_blocks; ++bid)
117     {
118         // estimating indices
119         starting_index = L*bid;
120         ending_index = std::min(starting_index + block_size - 1, maxlength -
121                                1);
122         length_left_to_fill = ending_index - starting_index;
123
124         // copying to the common-block
125         std::copy(large_vector.begin() + starting_index,
126                 large_vector.begin() + ending_index + 1,
127                 canvas.begin());
128
129         // performing convolution
130         block_conv_output = svr::conv1D_fftw(canvas,
131                                             small_vector);
132
133         // discarding edges and writing values
134         std::copy(block_conv_output.begin() + filter_size-2,

```

```

134         block_conv_output.begin() + filter_size-2 +
            std::min(static_cast<int>(L-1),
                    static_cast<int>(length_left_to_fill)) + 1,
135         finaloutput.begin()+starting_index);
136     }
137
138     // returning
139     return std::move(finaloutput);
140
141 }
142
143 /*=====
144 Long-signal Conv1D with FFT Plan
145 improvements:
146     > make an inplace version of this
147 -----*/
148 template <typename T>
149 auto conv1D_long_prototype(
150     const std::vector<T>& input_vector_A,
151     const std::vector<T>& input_vector_B,
152     svr::FFTPlanClass<T, std::complex<T>>& fft_plan,
153     svr::IFFTPlanClass<std::complex<T>, T>& ifft_plan
154 )
155 {
156     // Error checks
157     if (fft_plan.nfft_ != ifft_plan.nfft_)
158         throw std::runtime_error("fft_plan.nfft_ != ifft_plan.nfft_");
159
160     // fetching references to large-signal and small-signal
161     const auto& large_signal_original {
162         input_vector_A.size() >= input_vector_B.size() ?
163         input_vector_A : input_vector_B
164     };
165     const auto& small_signal {
166         input_vector_A.size() < input_vector_B.size() ?
167         input_vector_A : input_vector_B
168     };
169
170     // copying
171     auto large_signal {std::vector<double>(
172         input_vector_A.size() + input_vector_B.size() - 1
173     )};
174     std::copy(large_signal_original.begin(),
175         large_signal_original.end(),
176         large_signal.begin());
177
178     // calculating parameters
179     const auto signal_size {large_signal_original.size()};
180     const auto filter_size {small_signal.size()};
181     const auto input_signal_block_size {fft_plan.nfft_ + 1 - filter_size};
182     if (input_signal_block_size <= 0)
183         throw std::runtime_error("input_signal_block_size <= 0 ");
184     const auto block_output_length {fft_plan.nfft_};
185     const auto num_blocks {static_cast<int>(
186         1 + std::ceil((signal_size + filter_size - 2)/input_signal_block_size)
187     )};
188     const auto final_output_size {signal_size + filter_size - 1};
189     const auto useful_sample_length {block_output_length - (filter_size - 1)
190         - (filter_size - 1)};

```

```

190
191 // parameters for re-use
192 auto start_index      {static_cast<int>(0)};
193 auto end_index        {static_cast<int>(0)};
194 auto output_start_index {static_cast<int>(0)};
195
196 // calculating fft(filter)
197 auto filter_zero_padded {std::vector<double>(block_output_length, 0.0)};
198 std::copy(small_signal.begin(), small_signal.end(), filter_zero_padded.begin());
199 auto filter_FFT         {fft_plan.fft(filter_zero_padded)};
200
201 // allocating space for storing input-blocks
202 auto signal_block_zero_padded {std::vector<double>(block_output_length, 0.0)};
203 auto fftw_output              {std::vector<double>()};
204 auto conv_output              {std::vector<double>()};
205 auto finaloutput              {std::vector<double>(final_output_size, 0.0)};
206
207 // going through the values
208 svr::Timer timer("fft-loop");
209 for(auto i = 0; i<num_blocks; ++i){
210
211     // calculating bounds
212     auto analytical_start {
213         (i*static_cast<int>(input_signal_block_size)) -
214         (static_cast<int>(filter_size) - 1)
215     };
216     auto analytical_end    {(i+1)*input_signal_block_size - 1};
217     start_index = std::max(
218         static_cast<int>(0), static_cast<int>(analytical_start)
219     );
220     end_index = std::min(
221         static_cast<int>(signal_size-1), static_cast<int>(analytical_end)
222     ); // [start-index, end-index)
223
224     // copying values
225     signal_block_zero_padded = std::move(std::vector<double>(block_output_length,
226         0.0));
227     std::copy(large_signal.begin() + start_index,
228         large_signal.begin() + end_index + 1,
229         signal_block_zero_padded.begin() + start_index - analytical_start);
230
231     // performing ifft(fft(x) * fft(y))
232     fftw_output = ifft_plan.ifft(
233         fft_plan.fft(signal_block_zero_padded) * filter_FFT
234     );
235
236     // trimming away the first parts (since partial)
237     conv_output = std::vector<double>(fftw_output.begin() + filter_size -
238         1, fftw_output.end());
239
240     // writing to final-output
241     std::copy(conv_output.begin(), conv_output.end(), finaloutput.begin() +
242         output_start_index);
243     output_start_index += conv_output.size();
244 }
245
246 }
247
248 /*=====

```

```

245 Long-signal Conv1D with FFT-Plan-Pool
246 -----*/
247 template <typename T>
248 auto conv1D_long_FFTPlanPool(
249     const std::vector<T>& input_vector_A,
250     const std::vector<T>& input_vector_B,
251     svr::FFTPlanUniformPool<double, std::complex<double>>& uniform_pool
252 )
253 {
254
255     //
256
257     //
258
259     cout << format("aight, calm down\n");
260 }
261
262 /*=====
263 Short-conv1D
264 -----*/
265 // template <std::size_t shortsize,
266 //         typename T1,
267 //         typename T2>
268 template <typename T1,
269         typename T2>
270 auto conv1D_short(const std::vector<T1>& input_vector_A,
271                 const std::vector<T2>& input_vector_B)
272 {
273     // resulting type
274     using T3 = decltype(std::declval<T1>() * std::declval<T2>());
275
276     // creating canvas
277     auto canvas_length {input_vector_A.size() + input_vector_B.size() - 1};
278
279     // calculating fft of two arrays
280     auto fft_A {svr::fft(input_vector_A, canvas_length)};
281     auto fft_B {svr::fft(input_vector_B, canvas_length)};
282
283     // element-wise multiplying the two matrices
284     auto fft_AB {fft_A * fft_B};
285
286     // finding inverse FFT
287     auto convolved_result {ifft(fft_AB)};
288
289     // returning
290     // return std::move(convolved_result);
291     return convolved_result;
292 }
293
294
295
296 /*=====
297 1D Convolution of a matrix and a vector
298 -----*/
299 template <typename T>
300 auto conv1D(const std::vector<std::vector<T>>& input_matrix,
301            const std::vector<T>& input_vector,
302            const std::size_t& dim)
303 {

```

```

304 // getting dimensions
305 const auto& num_rows_matrix {input_matrix.size()};
306 const auto& num_cols_matrix {input_matrix[0].size()};
307 const auto& num_elements_vector {input_vector.size()};
308
309 // creating canvas
310 auto canvas {std::vector<std::vector<T>>()};
311
312 // creating output based on dim
313 if (dim == 1)
314 {
315     // performing convolutions row by row
316     for(auto row = 0; row < num_rows_matrix; ++row)
317     {
318         cout << format("\t\t row = {}/{}\n", row, num_rows_matrix);
319         auto bruh {conv1D(input_matrix[row], input_vector)};
320         auto bruh_real {svr::real(std::move(bruh))};
321
322         canvas.push_back(
323             svr::real(
324                 std::move(bruh_real)
325             )
326         );
327     }
328 }
329 else{
330     std::cerr << "svr_conv.hpp | conv1D | yet to be implemented \n";
331 }
332
333 // returning
334 return std::move(canvas);
335
336 }
337
338 /*=====
339 1D Convolution of a matrix and a vector (in-place)
340 -----*/
341
342 }

```

A.6 Coordinate Change

```

1 #pragma once
2 namespace svr {
3     /*=====
4     y = cart2sph(vector)
5     -----*/
6     template <typename T>
7     auto cart2sph(const std::vector<T>& cartesian_vector){
8
9         // splatting the point onto xy-plane
10        auto xysplat {cartesian_vector};
11        xysplat[2] = 0;
12
13        // finding splat lengths
14        auto xysplat_lengths {norm(xysplat)};

```

```

15
16 // finding azimuthal and elevation angles
17 auto azimuthal_angles {svr::atan2(xysplat[1],
18                                   xysplat[0]) \
19                                   * 180.00/std::numbers::pi};
20 auto elevation_angles {svr::atan2(cartesian_vector[2],
21                                   xysplat_lengths) \
22                                   * 180.00/std::numbers::pi};
23 auto rho_values {norm(cartesian_vector)};
24
25 // creating tensor to send back
26 auto spherical_vector {std::vector<T>{azimuthal_angles,
27                                       elevation_angles,
28                                       rho_values}};
29
30 // moving it back
31 return std::move(spherical_vector);
32 }
33 /*=====
34 y = cart2sph(vector)
35 -----*/
36 template <typename T>
37 auto cart2sph_inplace(std::vector<T>& cartesian_vector){
38
39 // splatting the point onto xy-plane
40 auto xysplat {cartesian_vector};
41 xysplat[2] = 0;
42
43 // finding splat lengths
44 auto xysplat_lengths {norm(xysplat)};
45
46 // finding azimuthal and elevation angles
47 auto azimuthal_angles {svr::atan2(xysplat[1], xysplat[0]) *
48                               180.00/std::numbers::pi};
49 auto elevation_angles {svr::atan2(cartesian_vector[2],
50                                   xysplat_lengths) * 180.00/std::numbers::pi};
51 auto rho_values {norm(cartesian_vector)};
52
53 // creating tesnor
54 cartesian_vector[0] = azimuthal_angles;
55 cartesian_vector[1] = elevation_angles;
56 cartesian_vector[2] = rho_values;
57 }
58 /*=====
59 y = cart2sph(input_matrix, dim)
60 -----*/
61 template <typename T>
62 auto cart2sph(const std::vector<std::vector<T>>& input_matrix,
63               const std::size_t axis)
64 {
65 // fetching dimensions
66 const auto& num_rows {input_matrix.size()};
67 const auto& num_cols {input_matrix[0].size()};
68
69 // checking the axis and dimensions
70 if (axis == 0 && num_rows != 3) {std::cerr << "cart2sph: incorrect num-elements
71 \n";}
72 if (axis == 1 && num_cols != 3) {std::cerr << "cart2sph: incorrect num-elements
73 \n";}

```



```

71
72 // creating canvas
73 auto canvas {std::vector<std::vector<T>>(
74     num_rows,
75     std::vector<T>(num_cols, 0)
76 )};
77
78 // if axis = 0, performing operation column-wise
79 if(axis == 0)
80 {
81     for(auto col = 0; col < num_cols; ++col)
82     {
83         // fetching current column
84         auto curr_column {std::vector<T>({input_matrix[0][col],
85                                           input_matrix[1][col],
86                                           input_matrix[2][col]})};
87
88         // performing inplace transformation
89         cart2sph_inplace(curr_column);
90
91         // storing it back
92         canvas[0][col] = curr_column[0];
93         canvas[1][col] = curr_column[1];
94         canvas[2][col] = curr_column[2];
95     }
96 }
97 // if axis == 1, performing operations row-wise
98 else if(axis == 0)
99 {
100     std::cerr << "cart2sph: yet to be implemented \n";
101 }
102 else
103 {
104     std::cerr << "cart2sph: yet to be implemented \n";
105 }
106
107 // returning
108 return std::move(canvas);
109
110 }
111
112 // =====
113 template <typename T>
114 auto sph2cart(const std::vector<T> spherical_vector){
115
116     // creating cartesian vector
117     auto cartesian_vector {std::vector<T>(spherical_vector.size(), 0)};
118
119     // populating
120     cartesian_vector[0] = spherical_vector[2] * \
121         cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
122         cos(spherical_vector[0] * std::numbers::pi / 180.00);
123     cartesian_vector[1] = spherical_vector[2] * \
124         cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
125         sin(spherical_vector[0] * std::numbers::pi / 180.00);
126     cartesian_vector[2] = spherical_vector[2] * \
127         sin(spherical_vector[1] * std::numbers::pi / 180.00);
128
129     // returning

```

```

130     return std::move(cartesian_vector);
131 }
132 }

```

A.7 Cosine

```

1  #pragma once
2  /*=====
3  y = cos(input_vector)
4  -----*/
5  template <typename T>
6  auto cos(const std::vector<T>& input_vector)
7  {
8      // created canvas
9      auto canvas {input_vector};
10
11     // calling the function
12     std::transform(input_vector.begin(), input_vector.end(),
13                   canvas.begin(),
14                   [](auto& argx){return std::cos(argx);});
15
16     // returning the output
17     return std::move(canvas);
18 }
19 /*=====
20 y = cosd(input_vector)
21 -----*/
22 template <typename T>
23 auto cosd(const std::vector<T> input_vector)
24 {
25     // created canvas
26     auto canvas {input_vector};
27
28     // calling the function
29     std::transform(input_vector.begin(),
30                   input_vector.end(),
31                   input_vector.begin(),
32                   [](const auto& argx){return std::cos(argx * 180.00/std::numbers::pi);});
33
34     // returning the output
35     return std::move(canvas);
36 }

```

A.8 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  };

```

```

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.9 Editing Index Values

```

1 #pragma once
2 /*=====
3 Matlab's equivalent of A[A < 0.5] = 0
4 -----*/
5 // template <typename T, typename BooleanVector, typename U>
6 // auto edit(std::vector<T>&          input_vector,
7 //           BooleanVector          bool_vector,
8 //           const    U             scalar)
9 // {
10 //     // throwing an error
11 //     if (input_vector.size() != bool_vector.size())
12 //         std::cerr << "edit: incompatible size\n";
13
14 //     // overwriting input-vector
15 //     std::transform(input_vector.begin(), input_vector.end(),
16 //                   bool_vector.begin(),
17 //                   input_vector.begin(),
18 //                   [&scalar](auto& argx, auto argy){
19 //                       if(argy == true) {return static_cast<T>(scalar);}
20 //                       else             {return argx;}
21 //                   });
22
23 //     // no-returns since in-place
24 // }
25 template <typename T, typename U>
26 auto edit(std::vector<T>&          input_vector,
27           const std::vector<bool>& bool_vector,
28           const    U             scalar)
29 {
30     // throwing an error
31     if (input_vector.size() != bool_vector.size())
32         std::cerr << "edit: incompatible size\n";
33
34     // overwriting input-vector
35     std::transform(input_vector.begin(), input_vector.end(),
36                   bool_vector.begin(),
37                   input_vector.begin(),
38                   [&scalar](auto& argx, auto argy){
39                       if(argy == true) {return static_cast<T>(scalar);}
40                       else             {return argx;}
41                   });
42
43     // no-returns since in-place
44 }

```

```

45
46  /*=====
47  accumulate version of edit, instead of just placing values
48
49  THings to add
50      - ensuring template only accepts int, std::size_t and similar for T2
51      - bring in histogram method to ensure SIMD
52  -----*/
53  template <typename T1,
54            typename T2>
55  auto edit_accumulate(std::vector<T1>& input_vector,
56                      const std::vector<T2>& indices_to_edit,
57                      const std::vector<T1>& new_values)
58  {
59      // certain checks
60      if (indices_to_edit.size() != new_values.size())
61          std::cerr << "svr::edit | edit_accumulate | size-disparity occurred \n";
62
63      // going through each and accumulating
64      for(auto i = 0; i < input_vector.size(); ++i){
65          const auto target_index {static_cast<std::size_t>(indices_to_edit[i])}; //
66          const auto new_value {new_values[i]};
67          input_vector[target_index] += new_value;
68      }
69
70      // no-return since in-place
71  }

```

A.10 Equality

```

1  #pragma once
2  /*=====
3  -----*/
4  template <typename T, typename U>
5  auto operator==(const std::vector<T>& input_vector,
6                 const U& scalar)
7  {
8      // setting up canvas
9      auto canvas {std::vector<bool>(input_vector.size())};
10
11      // writing to canvas
12      std::transform(input_vector.begin(), input_vector.end(),
13                    canvas.begin(),
14                    [&scalar](const auto& argx){
15                        return argx == scalar;
16                    });
17
18      // returning
19      return std::move(canvas);
20  }

```

A.11 Exponentiate

```

1  #pragma once

```

```

2  /*=====
3  y = abs(vector)
4  -----*/
5  template <typename T>
6  auto exp(const std::vector<T>& input_vector)
7  {
8      // creating canvas
9      auto canvas {input_vector};
10
11     // transforming
12     std::transform(canvas.begin(), canvas.end(),
13                   canvas.begin(),
14                   [](auto& argx){return std::exp(argx);});
15
16     // returning
17     return std::move(canvas);
18 }

```

A.12 FFT

```

1  #pragma once
2  namespace svr {
3      /*=====
4      For type-deductions
5      -----*/
6      template <typename T>
7      struct fft_result_type;
8
9      // specializations
10     template <> struct fft_result_type<double>{
11         using type = std::complex<double>;
12     };
13     template <> struct fft_result_type<std::complex<double>>{
14         using type = std::complex<double>;
15     };
16     template <> struct fft_result_type<float>{
17         using type = std::complex<float>;
18     };
19     template <> struct fft_result_type<std::complex<float>>{
20         using type = std::complex<float>;
21     };
22
23     template <typename T>
24     using fft_result_t = typename fft_result_type<T>::type;
25
26     /*=====
27     y = fft(x, nfft)
28     > calculating n-point dft where n-value is explicit
29     -----*/
30     template<typename T>
31     auto fft(const std::vector<T>& input_vector,
32             const size_t nfft)
33     {
34         // throwing an error
35         if (nfft < input_vector.size()) {std::cerr << "size-mismatch\n";}
36         if (nfft <= 0) {std::cerr << "size-mismatch\n";}

```

```

37
38 // fetching data-type
39 using RType = fft_result_t<T>;
40 using baseType = std::conditional_t<std::is_same_v<T, std::complex<double>>,
41                                     double,
42                                     T>;
43
44 // canvas instantiation
45 std::vector<RType> canvas(nfft);
46 auto nfft_sqrt = {static_cast<RType>(std::sqrt(nfft))};
47 auto finaloutput = {std::vector<RType>(nfft, 0)};
48
49 // calculating index by index
50 for(int frequency_index = 0; frequency_index < nfft; ++frequency_index){
51     RType accumulate_value;
52     for(int signal_index = 0; signal_index < input_vector.size(); ++signal_index){
53         accumulate_value += \
54             static_cast<RType>(input_vector[signal_index]) * \
55             static_cast<RType>(std::exp(-1.00 * std::numbers::pi * \
56                                     (static_cast<baseType>(frequency_index)/static_cast<baseType>(nfft) * \
57                                     * \
58                                     static_cast<baseType>(signal_index))));
59     }
60     finaloutput[frequency_index] = accumulate_value / nfft_sqrt;
61 }
62
63 // returning
64 return std::move(finaloutput);
65 }
66
67 /*=====
68 y = fft(std::vector<double> nfft) // specialization
69 -----*/
70 #include <fftw3.h> // for fft
71 template <>
72 auto fft(const std::vector<double>& input_vector,
73          const std::size_t nfft)
74 {
75     if (nfft < input_vector.size())
76         throw std::runtime_error("nfft must be >= input_vector.size()");
77     if (nfft <= 0)
78         throw std::runtime_error("nfft must be > 0");
79
80     // FFTW real-to-complex output
81     std::vector<std::complex<double>> output(nfft);
82
83     // Allocate input (double) and output (fftw_complex) arrays
84     double* in = reinterpret_cast<double*>(
85         fftw_malloc(sizeof(double) * nfft)
86     );
87     fftw_complex* out = reinterpret_cast<fftw_complex*>(
88         fftw_malloc(sizeof(fftw_complex) * (nfft/2 + 1))
89     );
90
91     // Copy input and zero-pad if needed
92     for (std::size_t i = 0; i < nfft; ++i) {
93         in[i] = (i < input_vector.size()) ? input_vector[i] : 0.0;
94     }

```

```

95     // Create FFTW plan and execute
96     fftw_plan plan = fftw_plan_dft_r2c_1d(
97         static_cast<int>(nfft), in, out, FFTW_ESTIMATE
98     );
99     fftw_execute(plan);
100
101     // Copy FFTW output to std::vector<std::complex<double>>
102     for (std::size_t i = 0; i < nfft/2 + 1; ++i) {
103         output[i] = std::complex<double>(out[i][0], out[i][1]);
104     }
105     // Optional: fill remaining bins with zeros to match full nfft size
106     for (std::size_t i = nfft/2 + 1; i < nfft; ++i) {
107         output[i] = std::complex<double>(0.0, 0.0);
108     }
109
110     // Cleanup
111     fftw_destroy_plan(plan);
112     fftw_free(in);
113     fftw_free(out);
114
115     // filling up the other half of the output
116     const auto halfpoint {static_cast<std::size_t>(nfft/2)};
117     std::transform(
118         output.begin() + 1,          // first half (skip DC)
119         output.begin() + halfpoint, // end of first half
120         output.rbegin(),             // start writing from last element backward (skip
            Nyquist)
121         [](const auto& x) { return std::conj(x); }
122     );
123
124     // returning
125     return std::move(output);
126 }
127
128
129 /*=====
130 y = ifft(x, nfft)
131 -----*/
132
133 template<typename T>
134 auto ifft(const std::vector<T>& input_vector)
135 {
136     // fetching data-type
137     using RType = fft_result_t<T>;
138     using baseType = std::conditional_t<std::is_same_v<T, std::complex<double>>,
139                                         double,
140                                         T>;
141
142     // setup
143     auto nfft {input_vector.size()};
144
145     // canvas instantiation
146     std::vector<RType> canvas(nfft);
147     auto nfft_sqrt {static_cast<RType>(std::sqrt(nfft))};
148     auto finaloutput {std::vector<RType>(nfft, 0)};
149
150     // calculating index by index
151     for(int frequency_index = 0; frequency_index<nfft; ++frequency_index){
152         RType accumulate_value;
153         for(int signal_index = 0; signal_index < input_vector.size(); ++signal_index){

```

```

153         accumulate_value += \
154             static_cast<RType>(input_vector[signal_index]) * \
155             static_cast<RType>(std::exp(1.00 * std::numbers::pi * \
156                 (static_cast<baseType>(frequency_index)/static_cast<baseType>(nfft)
157                     * \
158                     static_cast<baseType>(signal_index))));
159     }
160     finaloutput[frequency_index] = accumulate_value / nfft_sqrt;
161 }
162
163 // returning
164 return std::move(finaloutput);
165 }
166
167 /*=====
168 y = ifft()
169 using fftw
170 -----*/
171 // #include <fftw3.h>
172 // #include <vector>
173 // #include <complex>
174 // #include <stdexcept>
175 // #include <algorithm>
176
177 // // template <typename T>
178 // auto ifft(const std::vector<std::complex<double>>& input_vector,
179 //           const std::size_t nfft)
180 // {
181 //     if (nfft <= 0)
182 //         throw std::runtime_error("nfft must be > 0");
183 //     if (input_vector.size() < nfft/2 + 1)
184 //         throw std::runtime_error("input spectrum must have at least nfft/2+1
185 //             bins");
186
187 //     // Output: real-valued time-domain signal
188 //     std::vector<double> output(nfft);
189
190 //     // Allocate FFTW input/output
191 //     fftw_complex* in =
192 //         reinterpret_cast<fftw_complex*>(fftw_malloc(sizeof(fftw_complex) * (nfft/2 + 1)));
193 //     double* out = reinterpret_cast<double*>(fftw_malloc(sizeof(double) * nfft));
194
195 //     // Copy input spectrum into FFTW buffer
196 //     for (std::size_t i = 0; i < nfft/2 + 1; ++i) {
197 //         in[i][0] = input_vector[i].real();
198 //         in[i][1] = input_vector[i].imag();
199 //     }
200
201 //     // Create inverse plan and execute
202 //     fftw_plan plan = fftw_plan_dft_c2r_1d(static_cast<int>(nfft), in, out,
203 //         FFTW_ESTIMATE);
204 //     fftw_execute(plan);
205
206 //     // Normalize by nfft (FFTW does unscaled IFFT)
207 //     for (std::size_t i = 0; i < nfft; ++i) {
208 //         output[i] = out[i] / static_cast<double>(nfft);
209 //     }
210
211 //     // Cleanup

```



```

208 //     fftw_destroy_plan(plan);
209 //     fftw_free(in);
210 //     fftw_free(out);
211
212 //     return output;
213 // }
214
215 /*=====
216 x = ifft(std::vector<std::complex<double>> spectrum, nfft)
217 -----*/
218 #include <fftw3.h>
219 #include <vector>
220 #include <complex>
221 #include <stdexcept>
222
223 auto ifft(const std::vector<std::complex<double>>& input_vector,
224          const std::size_t nfft)
225 {
226     if (nfft <= 0)
227         throw std::runtime_error("nfft must be > 0");
228     if (input_vector.size() != nfft)
229         throw std::runtime_error("input spectrum must be of size nfft");
230
231     // Output: real-valued time-domain sequence
232     std::vector<double> output(nfft);
233
234     // Allocate FFTW input/output
235     fftw_complex* in = reinterpret_cast<fftw_complex*>(
236         fftw_malloc(sizeof(fftw_complex) * (nfft/2 + 1))
237     );
238     double* out = reinterpret_cast<double*>(
239         fftw_malloc(sizeof(double) * nfft)
240     );
241
242     // Copy *only* the first nfft/2+1 bins (rest are redundant due to symmetry)
243     for (std::size_t i = 0; i < nfft/2 + 1; ++i) {
244         in[i][0] = input_vector[i].real();
245         in[i][1] = input_vector[i].imag();
246     }
247
248     // Create inverse FFTW plan
249     fftw_plan plan = fftw_plan_dft_c2r_1d(
250         static_cast<int>(nfft),
251         in,
252         out,
253         FFTW_ESTIMATE
254     );
255
256     fftw_execute(plan);
257
258     // Normalize by nfft (FFTW leaves IFFT unscaled)
259     for (std::size_t i = 0; i < nfft; ++i) {
260         output[i] = out[i] / static_cast<double>(nfft);
261     }
262
263     // Cleanup
264     fftw_destroy_plan(plan);
265     fftw_free(in);
266     fftw_free(out);

```

```

267
268     return output;
269 }
270
271
272
273 }

```

A.13 Flipping Containers

```

1  #pragma once
2  namespace svr {
3      /*=====
4      Mirror image of a vector
5      -----*/
6      template <typename T>
7      auto fliplr(const std::vector<T>& input_vector)
8      {
9          // creating canvas
10         auto canvas {input_vector};
11
12         // rewriting
13         std::reverse(canvas.begin(), canvas.end());
14
15         // returning
16         return std::move(canvas);
17     }
18 }

```

A.14 Indexing

```

1  #pragma once
2  namespace svr {
3      /*=====
4      y = index(vector, mask)
5      -----*/
6      template <typename T1,
7                typename T2,
8                typename = std::enable_if_t<std::is_arithmetic_v<T1> ||
9                std::is_same_v<T1, std::complex<float>> > ||
10                std::is_same_v<T1, std::complex<double>> >
11                >
12                >
13      auto index(const std::vector<T1>& input_vector,
14                const std::vector<T2>& indices_to_sample)
15      {
16          // creating canvas
17          auto canvas {std::vector<T1>(indices_to_sample.size(), 0)};
18
19          // copying the associated values
20          for(int i = 0; i < indices_to_sample.size(); ++i){
21              auto source_index {indices_to_sample[i]};
22              if(source_index < input_vector.size()){
23                  canvas[i] = input_vector[source_index];

```

```

24     }
25     else
26         cout << "svr::index | source_index !< input_vector.size()\n";
27 }
28
29 // returning
30 return std::move(canvas);
31 }
32 /*=====
33 y = index(matrix, mask, dim)
34 -----*/
35 template <typename T1, typename T2>
36 auto index(const std::vector<std::vector<T1>>& input_matrix,
37           const std::vector<T2>& indices_to_sample,
38           const std::size_t& dim)
39 {
40     // fetching dimensions
41     const auto& num_rows_matrix {input_matrix.size()};
42     const auto& num_cols_matrix {input_matrix[0].size()};
43
44     // creating canvas
45     auto canvas {std::vector<std::vector<T1>>()};
46
47     // if indices are row-indices
48     if (dim == 0){
49
50         // initializing canvas
51         canvas = std::vector<std::vector<T1>>(
52             num_rows_matrix,
53             std::vector<T1>(indices_to_sample.size())
54         );
55
56         // filling the canvas
57         auto destination_index {0};
58         std::for_each(indices_to_sample.begin(), indices_to_sample.end(),
59             [&](const auto& col){
60                 for(auto row = 0; row < num_rows_matrix; ++row)
61                     canvas[row][destination_index] = input_matrix[row][col];
62                 ++destination_index;
63             });
64     }
65     else if(dim == 1){
66         // initializing canvas
67         canvas = std::vector<std::vector<T1>>(
68             indices_to_sample.size(),
69             std::vector<T1>(num_cols_matrix)
70         );
71
72         // filling the canvas
73         #pragma omp parallel for
74         for(auto row = 0; row < canvas.size(); ++row){
75             auto destination_col {0};
76             std::for_each(indices_to_sample.begin(), indices_to_sample.end(),
77                 [&row,
78                 &input_matrix,
79                 &destination_col,
80                 &canvas](const auto& source_col){
81                 canvas[row][destination_col++] =
                     input_matrix[row][source_col];

```

```

82         });
83     }
84 }
85 else {
86     std::cerr << "svr_index | this dim is not implemented \n";
87 }
88
89 // moving it back
90 return std::move(canvas);
91 }
92 }

```

A.15 Linspace

```

1  /*=====
2  Dependencies
3  -----*/
4  #pragma once
5  #include <vector>
6  #include <complex>
7
8
9  /*=====
10 in-place
11 -----*/
12 template <typename T>
13 auto linspace(auto& input,
14               const auto startvalue,
15               const auto endvalue,
16               const 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) {input[i] = startvalue + i*stepsize;}
20 };
21 /*=====
22 in-place
23 -----*/
24 template <typename T>
25 auto linspace(std::vector<std::complex<T>>& input,
26               const auto startvalue,
27               const auto endvalue,
28               const auto numpoints) -> void
29 {
30     auto stepsize = static_cast<T>(endvalue - startvalue)/static_cast<T>(numpoints-1);
31     for(int i = 0; i<input.size(); ++i) {
32         input[i] = startvalue + static_cast<T>(i)*stepsize;
33     }
34 };
35 /*=====
36 -----*/
37 template <typename T>
38 auto linspace(const T startvalue,
39               const T endvalue,
40               const std::size_t numpoints)
41 {
42     std::vector<T> input(numpoints);

```

```

43     auto stepsize = static_cast<T>(endvalue - startvalue)/static_cast<T>(numpoints-1);
44     for(int i = 0; i<input.size(); ++i) {input[i] = startvalue +
45         static_cast<T>(i)*stepsize;}
46     return std::move(input);
47 };
48 /*=====
49 -----*/
49 template <typename T, typename U>
50 auto linspace(const T startvalue,
51               const U endvalue,
52               const std::size_t numpoints)
53 {
54     std::vector<double> input(numpoints);
55     auto stepsize = static_cast<double>(endvalue -
56         startvalue)/static_cast<double>(numpoints-1);
57     for(int i = 0; i<input.size(); ++i) {input[i] = startvalue + i*stepsize;}
58     return std::move(input);
59 };

```

A.16 Max

```

1  #pragma once
2  /*=====
3  maximum along dimension 1
4  -----*/
5  template <std::size_t axis, typename T>
6  auto max(const std::vector<std::vector<T>>& input_matrix) -> std::enable_if_t<axis
7  == 1, std::vector<std::vector<T>>> >
8  {
9      // setting up canvas
10     auto canvas
11         {std::vector<std::vector<T>>(input_matrix.size(),std::vector<T>(1))};
12
13     // filling up the canvas
14     for(auto row = 0; row < input_matrix.size(); ++row)
15         canvas[row][0] = *(std::max_element(input_matrix[row].begin(),
16             input_matrix[row].end()));
17
18     // returning
19     return std::move(canvas);
20 }

```

A.17 Meshgrid

```

1  /*=====
2  Dependencies
3  -----*/
4  #pragma once
5  #include <vector> // for std::vector
6  #include <utility> // for std::pair
7  #include <complex> // for std::complex
8
9
10 /*=====

```

```

11 mesh-grid when working with l-values
12 -----*/
13 template <typename T>
14 auto meshgrid(const std::vector<T>& x,
15               const std::vector<T>& y)
16 {
17
18     // creating and filling x-grid
19     std::vector<std::vector<T>> xcanvas(y.size(), std::vector<T>(x.size(), 0));
20     for(auto row = 0; row < y.size(); ++row)
21         std::copy(x.begin(), x.end(), xcanvas[row].begin());
22
23     // creating and filling y-grid
24     std::vector<std::vector<T>> ycanvas(y.size(), std::vector<T>(x.size(), 0));
25     for(auto col = 0; col < x.size(); ++col)
26         for(auto row = 0; row < y.size(); ++row)
27             ycanvas[row][col] = y[row];
28
29     // returning
30     return std::move(std::pair{xcanvas, ycanvas});
31
32 }
33 /*=====
34 meshgrid when working with r-values
35 -----*/
36 template <typename T>
37 auto meshgrid(std::vector<T>&& x,
38               std::vector<T>&& y)
39 {
40
41     // creating and filling x-grid
42     std::vector<std::vector<T>> xcanvas(y.size(), std::vector<T>(x.size(), 0));
43     for(auto row = 0; row < y.size(); ++row)
44         std::copy(x.begin(), x.end(), xcanvas[row].begin());
45
46     // creating and filling y-grid
47     std::vector<std::vector<T>> ycanvas(y.size(), std::vector<T>(x.size(), 0));
48     for(auto col = 0; col < x.size(); ++col)
49         for(auto row = 0; row < y.size(); ++row)
50             ycanvas[row][col] = y[row];
51
52     // returning
53     return std::move(std::pair{xcanvas, ycanvas});
54
55 }

```

A.18 Minimum

```

1 #pragma once
2 /*=====
3 minimum along dimension 1
4 -----*/
5 template <std::size_t axis, typename T>
6 auto min(const std::vector<std::vector<T>>& input_matrix) -> std::enable_if_t<axis ==
7     1, std::vector<std::vector<T>> >
8 {

```

```

8 // creating canvas
9 auto canvas
    {std::vector<std::vector<T>>(input_matrix.size(),std::vector<T>(1))};
10
11 // storing the values
12 for(auto row = 0; row < input_matrix.size(); ++row)
13     canvas[row][0] = *(std::min_element(input_matrix[row].begin(),
14                                         input_matrix[row].end()));
15
16 // returning the value
17 return std::move(canvas);
18 }

```

A.19 Norm

```

1 #pragma once
2 /*=====
3 calculating norm for vector
4 -----*/
5 template <typename T>
6 auto norm(const std::vector<T>& input_vector)
7 {
8     return std::sqrt(
9         std::inner_product(
10             input_vector.begin(), input_vector.end(),
11             input_vector.begin(),
12             (T)0
13         )
14     );
15 }
16 /*=====
17 Calculating norm of a complex-vector
18 -----*/
19 template <>
20 auto norm(const std::vector<std::complex<double>>& input_vector)
21 {
22     return std::sqrt(
23         std::inner_product(
24             input_vector.begin(), input_vector.end(),
25             input_vector.begin(),
26             static_cast<double>(0),
27             std::plus<double>(),
28             [](const auto& argx,
29               const auto& argy){
30                 return static_cast<double>(
31                     (argx * std::conj(argy)).real()
32                 );
33             }
34         )
35     );
36 }
37 }
38 /*=====
39 -----*/
40 template <typename T>
41 auto norm(const std::vector<std::vector<T>>& input_matrix,

```

```

42         const std::size_t          dim)
43 {
44     // creating canvas
45     auto canvas = {std::vector<std::vector<T>>>()};
46     const auto& num_rows_matrix = {input_matrix.size()};
47     const auto& num_cols_matrix = {input_matrix[0].size()};
48
49     // along dim 0
50     if(dim == 0)
51     {
52         // allocate canvas
53         canvas = std::vector<std::vector<T>>>(
54             1,
55             std::vector<T>(input_matrix[0].size())
56         );
57
58         // performing norm
59         auto accumulate_vector = {std::vector<T>(input_matrix[0].size())};
60
61         // going through each row
62         for(auto row = 0; row < num_rows_matrix; ++row)
63         {
64             std::transform(input_matrix[row].begin(), input_matrix[row].end(),
65                             accumulate_vector.begin(),
66                             accumulate_vector.begin(),
67                             [](const auto& argx, auto& argy){
68                                 return argx*argx + argy;
69                             });
70         }
71
72         // calculating element-wise square root
73         std::for_each(accumulate_vector.begin(), accumulate_vector.end(),
74                       [](auto& argx){
75                           argx = std::sqrt(argx);
76                       });
77
78         // moving to the canvas
79         canvas[0] = std::move(accumulate_vector);
80     }
81     else if (dim == 1)
82     {
83         // allocating space in the canvas
84         canvas = std::vector<std::vector<T>>>(
85             input_matrix[0].size(),
86             std::vector<T>(1, 0)
87         );
88
89         // going through each column
90         for(auto row = 0; row < num_cols_matrix; ++row){
91             canvas[row][0] = norm(input_matrix[row]);
92         }
93     }
94 }
95 else
96 {
97     std::cerr << "norm(matrix, dim): dimension operation not defined \n";
98 }
99
100 // returning

```



```

101     return std::move(canvas);
102 }
103
104
105
106 /*
107 Templates to create
108 - matrix and norm-axis
109 - axis instantiated std::vector<T>
110 */

```

A.20 Division

```

1  #pragma once
2  /*=====
3  element-wise division with scalars
4  -----*/
5  template <typename T>
6  auto operator/(const std::vector<T>& input_vector,
7                const T& input_scalar)
8  {
9      // creating canvas
10     auto canvas {input_vector};
11
12     // filling canvas
13     std::transform(canvas.begin(), canvas.end(),
14                   canvas.begin(),
15                   [&input_scalar](const auto& argx){
16                       return static_cast<double>(argx) /
17                          static_cast<double>(input_scalar);
18                   });
19
20     // returning value
21     return std::move(canvas);
22 }
23 /*=====
24 element-wise division with scalars
25 -----*/
26 template <typename T>
27 auto operator/=(const std::vector<T>& input_vector,
28                const T& input_scalar)
29 {
30     // creating canvas
31     auto canvas {input_vector};
32
33     // filling canvas
34     std::transform(canvas.begin(), canvas.end(),
35                   canvas.begin(),
36                   [&input_scalar](const auto& argx){
37                       return static_cast<double>(argx) /
38                          static_cast<double>(input_scalar);
39                   });
40
41     // returning value
42     return std::move(canvas);
43 }

```

```

42  /*=====
43  element-wise with matrix
44  -----*/
45  template <typename T>
46  auto operator/(const std::vector<std::vector<T>>& input_matrix,
47                const T scalar)
48  {
49      // fetching matrix-dimensions
50      const auto& num_rows_matrix {input_matrix.size()};
51      const auto& num_cols_matrix {input_matrix[0].size()};
52
53      // creating canvas
54      auto canvas {std::vector<std::vector<T>>>(
55          num_rows_matrix,
56          std::vector<T>(num_cols_matrix)
57      )};
58
59      // dividing with values
60      for(auto row = 0; row < num_rows_matrix; ++row){
61          std::transform(input_matrix[row].begin(), input_matrix[row].end(),
62                        canvas[row].begin(),
63                        [&scalar](const auto& argx){
64                            return argx/scalar;
65                        });
66      }
67
68      // returning values
69      return std::move(canvas);
70  }

```

A.21 Addition

```

1  #pragma once
2  /*=====
3  y = vector + vector
4  -----*/
5  template <typename T>
6  std::vector<T> operator+(const std::vector<T>& a,
7                          const std::vector<T>& b)
8  {
9      // Identify which is bigger
10     const auto& big = (a.size() > b.size()) ? a : b;
11     const auto& small = (a.size() > b.size()) ? b : a;
12
13     std::vector<T> result = big; // copy the bigger one
14
15     // Add elements from the smaller one
16     for (size_t i = 0; i < small.size(); ++i) {
17         result[i] += small[i];
18     }
19
20     return result;
21 }
22 /*=====
23 -----*/
24 // y = vector + vector

```

```

25 template <typename T>
26 std::vector<T>& operator+=(std::vector<T>& a,
27                           const std::vector<T>& b) {
28
29     const auto& small = (a.size() < b.size()) ? a : b;
30     const auto& big = (a.size() < b.size()) ? b : a;
31
32     // If b is bigger, resize 'a' to match
33     if (a.size() < b.size()) {a.resize(b.size());}
34
35     // Add elements
36     for (size_t i = 0; i < small.size(); ++i) {a[i] += b[i];}
37
38     // returning elements
39     return a;
40 }
41 // =====
42 // y = matrix + matrix
43 template <typename T>
44 std::vector<std::vector<T>>> operator+(const std::vector<std::vector<T>>& a,
45                                       const std::vector<std::vector<T>>& b)
46 {
47     // fetching dimensions
48     const auto& num_rows_A {a.size()};
49     const auto& num_cols_A {a[0].size()};
50     const auto& num_rows_B {b.size()};
51     const auto& num_cols_B {b[0].size()};
52
53     // choosing the three different metrics
54     if (num_rows_A != num_rows_B && num_cols_A != num_cols_B){
55         cout << format("a.dimensions = [{},{}, b.shape = [{},{}]\n",
56                       num_rows_A, num_cols_A,
57                       num_rows_B, num_cols_B);
58         std::cerr << "dimensions don't match\n";
59     }
60
61     // creating canvas
62     auto canvas {std::vector<std::vector<T>>>(
63         std::max(num_rows_A, num_rows_B),
64         std::vector<T>(std::max(num_cols_A, num_cols_B), (T)0.00)
65     )};
66
67     // performing addition
68     if (num_rows_A == num_rows_B && num_cols_A == num_cols_B){
69         for(auto row = 0; row < num_rows_A; ++row){
70             std::transform(a[row].begin(), a[row].end(),
71                           b[row].begin(),
72                           canvas[row].begin(),
73                           std::plus<T>());
74         }
75     }
76     else if(num_rows_A == num_rows_B){
77
78         // if number of columns are different, check if one of the cols are one
79         const auto min_num_cols {std::min(num_cols_A, num_cols_B)};
80         if (min_num_cols != 1) {std::cerr<< "Operator+: unable to broadcast\n";}
81         const auto max_num_cols {std::max(num_cols_A, num_cols_B)};
82
83         // using references to tag em differently

```

```

84     const auto& big_matrix    {num_cols_A > num_cols_B ? a : b};
85     const auto& small_matrix  {num_cols_A < num_cols_B ? a : b};
86
87     // Adding to canvas
88     for(auto row = 0; row < canvas.size(); ++row){
89         std::transform(big_matrix[row].begin(), big_matrix[row].end(),
90                        canvas[row].begin(),
91                        [&small_matrix,
92                         &row](const auto& argx){
93                             return argx + small_matrix[row][0];
94                         });
95     }
96 }
97 else if(num_cols_A == num_cols_B){
98
99     // check if the smallest column-number is one
100    const auto min_num_rows {std::min(num_rows_A, num_rows_B)};
101    if(min_num_rows != 1)    {std::cerr << "Operator+ : unable to broadcast\n";}
102    const auto max_num_rows {std::max(num_rows_A, num_rows_B)};
103
104    // using references to differentiate the two matrices
105    const auto& big_matrix    {num_rows_A > num_rows_B ? a : b};
106    const auto& small_matrix  {num_rows_A < num_rows_B ? a : b};
107
108    // adding to canvas
109    for(auto row = 0; row < canvas.size(); ++row){
110        std::transform(big_matrix[row].begin(), big_matrix[row].end(),
111                       small_matrix[0].begin(),
112                       canvas[row].begin(),
113                       [](const auto& argx, const auto& argy){
114                           return argx + argy;
115                       });
116    }
117 }
118 else {
119     std::cerr << "operator+: yet to be implemented \n";
120 }
121
122 // returning
123 return std::move(canvas);
124 }
125 /*=====
126 y = vector + scalar
127 -----*/
128 template <typename T>
129 auto operator+(const std::vector<T>& input_vector,
130               const T scalar)
131 {
132     // creating canvas
133     auto canvas {input_vector};
134
135     // adding scalar to the canvas
136     std::transform(canvas.begin(), canvas.end(),
137                    canvas.begin(),
138                    [&scalar](auto& argx){return argx + scalar;});
139
140     // returning canvas
141     return std::move(canvas);
142 }

```

```

143  /*=====
144  y = scalar + vector
145  -----*/
146  template <typename T>
147  auto operator+(const T scalar,
148                const std::vector<T>& input_vector)
149  {
150      // creating canvas
151      auto canvas {input_vector};
152
153      // adding scalar to the canvas
154      std::transform(canvas.begin(), canvas.end(),
155                    canvas.begin(),
156                    [&scalar](auto& argx){return argx + scalar;});
157
158      // returning canvas
159      return std::move(canvas);
160  }

```

A.22 Multiplication (Element-wise)

```

1  #pragma once
2  /*=====
3  y = scalar * vector
4  -----*/
5  template <typename T>
6  auto operator*(const T scalar,
7                const std::vector<T>& input_vector)
8  {
9      // creating canvas
10     auto canvas {input_vector};
11     // performing operation
12     std::for_each(canvas.begin(), canvas.end(),
13                   [&scalar](auto& argx){argx = argx * scalar;});
14     // returning
15     return std::move(canvas);
16 }
17 /*=====
18 y = scalar * vector
19 -----*/
20 template <typename T1, typename T2,
21         typename = std::enable_if_t!std::is_same_v<std::decay_t<T1>, std::vector<T2>>>>
22 auto operator*(const T1 scalar,
23               const vector<T2>& input_vector)
24 {
25     // fetching final-type
26     using T3 = decltype(std::declval<T1>() * std::declval<T2>());
27     // creating canvas
28     auto canvas {std::vector<T3>(input_vector.size())};
29     // multiplying
30     std::transform(input_vector.begin(), input_vector.end(),
31                   canvas.begin(),
32                   [&scalar](auto& argx){
33                       return static_cast<T3>(scalar) * static_cast<T3>(argx);
34                   });
35     // returning

```

```

36     return std::move(canvas);
37 }
38 /*=====
39 y = vector * scalar
40 -----*/
41 template <typename T>
42 auto operator*(const std::vector<T>& input_vector,
43               const T scalar)
44 {
45     // creating canvas
46     auto canvas {input_vector};
47     // multiplying
48     std::for_each(canvas.begin(), canvas.end(),
49                 [&scalar](auto& argx){
50         argx = argx * scalar;
51     });
52     // returning
53     return std::move(canvas);
54 }
55 /*=====
56 y = vector * vector
57 -----*/
58 template <typename T>
59 auto operator*(const std::vector<T>& input_vector_A,
60               const std::vector<T>& input_vector_B)
61 {
62     // throwing error: size-disparity
63     if (input_vector_A.size() != input_vector_B.size()) {std::cerr << "operator*: size
        disparity \n";}
64
65     // creating canvas
66     auto canvas {input_vector_A};
67
68     // element-wise multiplying
69     std::transform(input_vector_B.begin(), input_vector_B.end(),
70                 canvas.begin(),
71                 canvas.begin(),
72                 [](const auto& argx, const auto& argy){
73         return argx * argy;
74     });
75
76     // moving it back
77     return std::move(canvas);
78 }
79 /*=====
80 -----*/
81 template <typename T1, typename T2>
82 auto operator*(const std::vector<T1>& input_vector_A,
83               const std::vector<T2>& input_vector_B)
84 {
85
86     // checking size disparity
87     if (input_vector_A.size() != input_vector_B.size())
88         std::cerr << "operator*: error, size-disparity \n";
89
90     // figuring out resulting data type
91     using T3 = decltype(std::declval<T1>() * std::declval<T2>());
92
93     // creating canvas

```

```

94     auto    canvas    {std::vector<T3>(input_vector_A.size())};
95
96     // performing multiplications
97     std::transform(input_vector_A.begin(), input_vector_A.end(),
98                   input_vector_B.begin(),
99                   canvas.begin(),
100                   [](const auto&    argx,
101                     const auto&    argy){
102                         return static_cast<T3>(argx) * static_cast<T3>(argy);
103                     });
104
105     // returning
106     return std::move(canvas);
107
108 }
109 /*=====
110 -----*/
111 // scalar * matrix =====
112 template <typename T>
113 auto operator*(const T scalar,
114               const std::vector<std::vector<T>>& inputMatrix)
115 {
116     std::vector<std::vector<T>> temp {inputMatrix};
117     for(int i = 0; i<inputMatrix.size(); ++i){
118         std::transform(inputMatrix[i].begin(),
119                       inputMatrix[i].end(),
120                       temp[i].begin(),
121                       [&scalar](T x){return scalar * x;});
122     }
123     return std::move(temp);
124 }
125 /*=====
126 y = matrix * scalar
127 -----*/
128 template <typename T>
129 auto operator*(const std::vector<std::vector<T>>& input_matrix,
130               const T scalar)
131 {
132     // fetching matrix dimensions
133     const auto& num_rows_matrix {input_matrix.size()};
134     const auto& num_cols_matrix {input_matrix[0].size()};
135
136     // creating canvas
137     auto canvas {std::vector<std::vector<T>>(
138         num_rows_matrix,
139         std::vector<T>(num_cols_matrix)
140     )};
141
142     // storing the values
143     for(auto row = 0; row < num_rows_matrix; ++row)
144         std::transform(input_matrix[row].begin(), input_matrix[row].end(),
145                       canvas[row].begin(),
146                       [&scalar](const auto& argx){
147                           return argx * scalar;
148                       });
149
150     // returning
151     return std::move(canvas);
152 }

```

```

153  /*=====
154  y = matrix * matrix
155  -----*/
156  template <typename T>
157  auto operator*(const std::vector<std::vector<T>>& A,
158                const std::vector<std::vector<T>>& B) -> std::vector<std::vector<T>>
159  {
160      // Case 1: element-wise multiplication
161      if (A.size() == B.size() && A[0].size() == B[0].size()) {
162          std::vector<std::vector<T>> C(A.size(), std::vector<T>(A[0].size()));
163          for (std::size_t row = 0; row < A.size(); ++row) {
164              std::transform(A[row].begin(), A[row].end(),
165                            B[row].begin(),
166                            C[row].begin(),
167                            [](const auto& x, const auto& y){ return x * y; });
168          }
169          return C;
170      }
171
172      // Case 2: broadcast column vector
173      else if (A.size() == B.size() && B[0].size() == 1) {
174          std::vector<std::vector<T>> C(A.size(), std::vector<T>(A[0].size()));
175          for (std::size_t row = 0; row < A.size(); ++row) {
176              std::transform(A[row].begin(), A[row].end(),
177                            C[row].begin(),
178                            [&](const auto& x){ return x * B[row][0]; });
179          }
180          return C;
181      }
182
183      // case 3: when second matrix contains just one row
184      // case 4: when first matrix is just one column
185      // case 5: when second matrix is just one column
186
187      // Otherwise, invalid
188      else {
189          throw std::runtime_error("operator* dimension mismatch");
190      }
191  }
192  /*=====
193  y = scalar * matrix
194  -----*/
195  template <typename T1, typename T2>
196  auto operator*(const T1 scalar,
197                const std::vector<std::vector<T2>>& inputMatrix)
198  {
199      std::vector<std::vector<T2>> temp {inputMatrix};
200      for(int i = 0; i<inputMatrix.size(); ++i){
201          std::transform(inputMatrix[i].begin(),
202                        inputMatrix[i].end(),
203                        temp[i].begin(),
204                        [&scalar](T2 x){return static_cast<T2>(scalar) * x;});
205      }
206      return temp;
207  }
208  /*=====
209  matrix-multiplication
210  -----*/
211  template <typename T1, typename T2>

```



```

212 auto matmul(const std::vector<std::vector<T1>>& matA,
213             const std::vector<std::vector<T2>>& matB)
214 {
215
216     // throwing error
217     if (matA[0].size() != matB.size()) {std::cerr << "dimension-mismatch \n";}
218
219     // getting result-type
220     using ResultType = decltype(std::declval<T1>() * std::declval<T2>() + \
221                                std::declval<T1>() * std::declval<T2>() );
222
223     // creating aliases
224     auto finalnumrows {matA.size()};
225     auto finalnumcols {matB[0].size()};
226
227     // creating placeholder
228     auto rowcolproduct = [&](auto rowA, auto colB){
229         ResultType temp {0};
230         for(int i = 0; i < matA.size(); ++i) {temp +=
231             static_cast<ResultType>(matA[rowA][i]) +
232             static_cast<ResultType>(matB[i][colB]);}
233         return temp;
234     };
235
236     // producing row-column combinations
237     std::vector<std::vector<ResultType>> finaloutput(finalnumrows,
238         std::vector<ResultType>(finalnumcols));
239     for(int row = 0; row < finalnumrows; ++row){for(int col = 0; col < finalnumcols;
240         ++col){finaloutput[row][col] = rowcolproduct(row, col);}}
241
242     // returning
243     return finaloutput;
244 }
245
246 /*=====
247 y = matrix * vector
248 -----*/
249
250 template <typename T>
251 auto operator*(const std::vector<std::vector<T>> input_matrix,
252               const std::vector<T> input_vector)
253 {
254     // fetching dimensions
255     const auto& num_rows_matrix {input_matrix.size()};
256     const auto& num_cols_matrix {input_matrix[0].size()};
257     const auto& num_rows_vector {1};
258     const auto& num_cols_vector {input_vector.size()};
259
260     const auto& max_num_rows {num_rows_matrix > num_rows_vector ? \
261         num_rows_matrix : num_rows_vector};
262     const auto& max_num_cols {num_cols_matrix > num_cols_vector ? \
263         num_cols_matrix : num_cols_vector};
264
265     // creating canvas
266     auto canvas {std::vector<std::vector<T>>(
267         max_num_rows,
268         std::vector<T>(max_num_cols, 0)
269     )};
270
271     //
272     if (num_cols_matrix == 1 && num_rows_vector == 1){

```

```

267
268     // writing to canvas
269     for(auto row = 0; row < max_num_rows; ++row)
270         for(auto col = 0; col < max_num_cols; ++col)
271             canvas[row][col] = input_matrix[row][0] * input_vector[col];
272 }
273 else{
274     std::cerr << "Operator*: [matrix, vector] | not implemented \n";
275 }
276
277 // returning
278 return std::move(canvas);
279
280 }
281 /*=====
282 scalar operators
283 -----*/
284 auto operator*(const std::complex<double> complexscalar,
285               const double doublescalar){
286     return complexscalar * static_cast<std::complex<double>>(doublescalar);
287 }
288 auto operator*(const double doublescalar,
289               const std::complex<double> complexscalar){
290     return complexscalar * static_cast<std::complex<double>>(doublescalar);
291 }
292 auto operator*(const std::complex<double> complexscalar,
293               const int scalar){
294     return complexscalar * static_cast<std::complex<double>>(scalar);
295 }
296 auto operator*(const int scalar,
297               const std::complex<double> complexscalar){
298     return complexscalar * static_cast<std::complex<double>>(scalar);
299 }

```

A.23 Subtraction

```

1  #pragma once
2  /*=====
3  y = vector - scalar
4  -----*/
5  template <typename T>
6  auto operator-(const std::vector<T>& a,
7                const T scalar){
8      std::vector<T> temp(a.size());
9      std::transform(a.begin(),
10                   a.end(),
11                   temp.begin(),
12                   [scalar](T x){return (x - scalar);});
13     return std::move(temp);
14 }
15 /*=====
16 y = vector - vector
17 -----*/
18 template <typename T>
19 auto operator-(const std::vector<T>& input_vector_A,
20               const std::vector<T>& input_vector_B)

```

```

21 {
22     // throwing error
23     if (input_vector_A.size() != input_vector_B.size())
24         std::cerr << "operator-(vector, vector): size disparity\n";
25
26     // creating canvas
27     const auto& num_cols {input_vector_A.size()};
28     auto canvas {std::vector<T>{}};
29
30     // performing operations
31     std::transform(input_vector_A.begin(), input_vector_A.begin(),
32                   input_vector_B.begin(),
33                   canvas.begin(),
34                   [](const auto& argx, const auto& argy){
35                       return argx - argy;
36                   });
37
38     // return
39     return std::move(canvas);
40 }
41 /*=====
42 y = matrix - matrix
43 -----*/
44 template <typename T>
45 auto operator-(const std::vector<std::vector<T>>& input_matrix_A,
46               const std::vector<std::vector<T>>& input_matrix_B)
47 {
48     // fetching dimensions
49     const auto& num_rows_A {input_matrix_A.size()};
50     const auto& num_cols_A {input_matrix_A[0].size()};
51     const auto& num_rows_B {input_matrix_B.size()};
52     const auto& num_cols_B {input_matrix_B[0].size()};
53
54     // creating canvas
55     auto canvas {std::vector<std::vector<T>>{}};
56
57     // if both matrices are of equal dimensions
58     if (num_rows_A == num_rows_B && num_cols_A == num_cols_B)
59     {
60         // copying one to the canvas
61         canvas = input_matrix_A;
62
63         // subtracting
64         for(auto row = 0; row < num_rows_B; ++row)
65             std::transform(canvas[row].begin(), canvas[row].end(),
66                           input_matrix_B[row].begin(),
67                           canvas[row].begin(),
68                           [](auto& argx, const auto& argy){
69                               return argx - argy;
70                           });
71     }
72     // column broadcasting (case 1)
73     else if (num_rows_A == num_rows_B && num_cols_B == 1)
74     {
75         // copying canvas
76         canvas = input_matrix_A;
77
78         // subtracting
79         for(auto row = 0; row < num_rows_A; ++row){

```

```

80         std::transform(canvas[row].begin(), canvas[row].end(),
81                         canvas[row].begin(),
82                         [&input_matrix_B,
83                          &row](auto& argx){
84                             return argx - input_matrix_B[row][0];
85                         });
86     }
87 }
88 else{
89     std::cerr << "operator-: not implemented for this case \n";
90 }
91
92 // returning
93 return std::move(canvas);
94 }

```

A.24 Operator Overloadings

A.25 Printing Containers

```

1  #pragma once
2  /*=====
3  -----*/
4  template<typename T>
5  void fPrintVector(const vector<T> input){
6      for(auto x: input) cout << x << ",";
7      cout << endl;
8  }
9
10 template<typename T>
11 void fpv(vector<T> input){
12     for(auto x: input) cout << x << ",";
13     cout << endl;
14 }
15 /*=====
16 -----*/
17 template<typename T>
18 void fPrintMatrix(const std::vector<std::vector<T>> input_matrix){
19     for(const auto& row: input_matrix)
20         cout << format("{}\n", row);
21 }
22 /*=====
23 -----*/
24 template <typename T>
25 void fPrintMatrix(const string& input_string,
26                  const std::vector<std::vector<T>> input_matrix){
27     cout << format("{} = \n", input_string);
28     for(const auto& row: input_matrix)
29         cout << format("{}\n", row);
30 }
31 /*=====
32 -----*/
33 template<typename T, typename T1>

```

```

34 void fPrintHashMap(unordered_map<T, T1> input){
35     for(auto x: input){
36         cout << format("[{} , {}] | ", x.first, x.second);
37     }
38     cout << endl;
39 }
40 /*=====
41 -----*/
42 void fPrintBinaryTree(TreeNode* root){
43     // sending it back
44     if (root == nullptr) return;
45
46     // printing
47     PRINTLINE
48     cout << "root->val = " << root->val << endl;
49
50     // calling the children
51     fPrintBinaryTree(root->left);
52     fPrintBinaryTree(root->right);
53
54     // returning
55     return;
56 }
57 /*=====
58 -----*/
59 void fPrintLinkedList(ListNode* root){
60     if (root == nullptr) return;
61     cout << root->val << " -> ";
62     fPrintLinkedList(root->next);
63     return;
64 }
65 /*=====
66 -----*/
67 template<typename T>
68 void fPrintContainer(T input){
69     for(auto x: input) cout << x << ", ";
70     cout << endl;
71     return;
72 }
73 /*=====
74 -----*/
75 template <typename T>
76 auto size(std::vector<std::vector<T>> inputMatrix){
77     cout << format("[{} , {}]\n",
78         inputMatrix.size(),
79         inputMatrix[0].size());
80 }
81 /*=====
82 -----*/
83 template <typename T>
84 auto size(const std::string& inputstring,
85     const std::vector<std::vector<T>>& inputMatrix){
86     cout << format("{} = [{} , {}]\n",
87         inputstring,
88         inputMatrix.size(),
89         inputMatrix[0].size());
90 }
91 }

```

A.26 Random Number Generation

```

1  #pragma once
2  /*=====
3  -----*/
4  template <typename T>
5  auto rand(const T min,
6            const T max) {
7      static std::random_device rd; // Seed
8      static std::mt19937 gen(rd()); // Mersenne Twister generator
9      std::uniform_real_distribution<> dist(min, max);
10     return dist(gen);
11 }
12 /*=====
13 -----*/
14 template <typename T>
15 auto rand(const T min,
16           const T max,
17           const std::size_t numelements)
18 {
19     static std::random_device rd; // Seed
20     static std::mt19937 gen(rd()); // Mersenne Twister generator
21     std::uniform_real_distribution<> dist(min, max);
22
23     // building the finaloutput
24     vector<T> finaloutput(numelements);
25     for(int i = 0; i<finaloutput.size(); ++i) {finaloutput[i] =
26         static_cast<T>(dist(gen));}
27
28     return finaloutput;
29 }
30 /*=====
31 -----*/
32 template <typename T>
33 auto rand(const T argmin,
34           const T argmax,
35           const std::vector<int> dimensions)
36 {
37     // throwing an error if dimension is greater than two
38     if (dimensions.size() > 2) {std::cerr << "dimensions are too high\n";}
39
40     // creating random engine
41     static std::random_device rd; // Seed
42     static std::mt19937 gen(rd()); // Mersenne Twister generator
43     std::uniform_real_distribution<> dist(argmin, argmax);
44
45     // building the finaloutput
46     vector<vector<T>> finaloutput;
47     for(int i = 0; i<dimensions[0]; ++i){
48         vector<T> temp;
49         for(int j = 0; j<dimensions[1]; ++j) {temp.push_back(dist(gen));}
50         // cout << format("\t\t temp = {}\n", temp);
51
52         finaloutput.push_back(temp);
53     }
54
55     // returning the finaloutput
56     return finaloutput;

```

```

57 }
58 }
59 /*=====
60 -----*/
61 auto rand(const std::vector<int> dimensions)
62 {
63     using ReturnType = double;
64
65     // throwing an error if dimension is greater than two
66     if (dimensions.size() > 2) {std::cerr << "dimensions are too high\n";}
67
68     // creating random engine
69     static std::random_device rd; // Seed
70     static std::mt19937 gen(rd()); // Mersenne Twister generator
71     std::uniform_real_distribution<> dist(0.00, 1.00);
72
73     // building the finaloutput
74     vector<vector<ReturnType>>> finaloutput;
75     for(int i = 0; i<dimensions[0]; ++i){
76         vector<ReturnType> temp;
77         for(int j = 0; j<dimensions[1]; ++j) {temp.push_back(dist(gen));}
78         finaloutput.push_back(std::move(temp));
79     }
80
81     // returning the finaloutput
82     return std::move(finaloutput);
83 }
84 }
85 /*=====
86 -----*/
87 template <typename T>
88 auto rand_complex_double(const T argmin,
89                          const T argmax,
90                          const std::vector<int>& dimensions)
91 {
92
93     // throwing an error if dimension is greater than two
94     if (dimensions.size() > 2) {std::cerr << "dimensions are too high\n";}
95
96     // creating random engine
97     static std::random_device rd; // Seed
98     static std::mt19937 gen(rd()); // Mersenne Twister generator
99     std::uniform_real_distribution<> dist(argmin, argmax);
100
101     // building the finaloutput
102     vector<vector<complex<double>>>> finaloutput;
103     for(int i = 0; i<dimensions[0]; ++i){
104         vector<complex<double>> temp;
105         for(int j = 0; j<dimensions[1]; ++j)
106             {temp.push_back(static_cast<double>(dist(gen)));}
107         finaloutput.push_back(std::move(temp));
108     }
109
110     // returning the finaloutput
111     return finaloutput;
112 }

```

A.27 Reshape

```

1  #pragma once
2
3  /*=====
4  reshaping a matrix into another matrix
5  -----*/
6  template <std::size_t M, std::size_t N, typename T>
7  auto reshape(const std::vector<std::vector<T>>& input_matrix){
8
9      // verifying size stuff
10     if (M*N != input_matrix.size() * input_matrix[0].size())
11         std::cerr << "Dimensions are quite different\n";
12
13     // creating canvas
14     auto canvas {std::vector<std::vector<T>>(
15         M, std::vector<T>(N, (T)0)
16     )};
17
18     // writing to canvas
19     size_t tid {0};
20     size_t target_row {0};
21     size_t target_col {0};
22     for(auto row = 0; row<input_matrix.size(); ++row){
23         for(auto col = 0; col < input_matrix[0].size(); ++col){
24             tid = row * input_matrix[0].size() + col;
25             target_row = tid/N;
26             target_col = tid%N;
27             canvas[target_row][target_col] = input_matrix[row][col];
28         }
29     }
30
31     // moving it back
32     return std::move(canvas);
33 }
34 /*=====
35 reshaping a matrix into a vector
36 -----*/
37 template<std::size_t M, typename T>
38 auto reshape(const std::vector<std::vector<T>>& input_matrix){
39
40     // checking element-count validity
41     if (M != input_matrix.size() * input_matrix[0].size())
42         std::cerr << "Number of elements differ\n";
43
44     // creating canvas
45     auto canvas {std::vector<T>(M, 0)};
46
47     // filling canvas
48     for(auto row = 0; row < input_matrix.size(); ++row)
49         for(auto col = 0; col < input_matrix[0].size(); ++col)
50             canvas[row * input_matrix.size() + col] = input_matrix[row][col];
51
52     // moving it back
53     return std::move(canvas);
54 }
55 /*=====
56 Matrix to matrix
57 -----*/

```



```

58 template<typename T>
59 auto reshape(const std::vector<std::vector<T>>& input_matrix,
60             const std::size_t M,
61             const std::size_t N){
62
63     // checking element-count validity
64     if ( M * N != input_matrix.size() * input_matrix[0].size())
65         std::cerr << "Number of elements differ\n";
66
67     // creating canvas
68     auto canvas {std::vector<std::vector<T>>(
69         M, std::vector<T>(N, (T)0)
70     )};
71
72     // writing to canvas
73     size_t tid {0};
74     size_t target_row {0};
75     size_t target_col {0};
76     for(auto row = 0; row<input_matrix.size(); ++row){
77         for(auto col = 0; col < input_matrix[0].size(); ++col){
78             tid = row * input_matrix[0].size() + col;
79             target_row = tid/N;
80             target_col = tid%N;
81             canvas[target_row][target_col] = input_matrix[row][col];
82         }
83     }
84
85     // moving it back
86     return std::move(canvas);
87 }
88 /*=====
89 converting a matrix into a vector
90 -----*/
91 template<typename T>
92 auto reshape(const std::vector<std::vector<T>>& input_matrix,
93             const size_t M){
94
95     // checking element-count validity
96     if (M != input_matrix.size() * input_matrix[0].size())
97         std::cerr << "Number of elements differ\n";
98
99     // creating canvas
100    auto canvas {std::vector<T>(M, 0)};
101
102    // filling canvas
103    for(auto row = 0; row < input_matrix.size(); ++row)
104        for(auto col = 0; col < input_matrix[0].size(); ++col)
105            canvas[row * input_matrix.size() + col] = input_matrix[row][col];
106
107    // moving it back
108    return std::move(canvas);
109 }

```

A.28 Summing with containers

```
1 #pragma once
```

```

2  /*=====
3  -----*/
4  template <std::size_t axis, typename T>
5  auto sum(const std::vector<T>& input_vector) -> std::enable_if_t<axis == 0,
        std::vector<T>>
6  {
7      // returning the input as is
8      return input_vector;
9  }
10 /*=====
11 -----*/
12 template <std::size_t axis, typename T>
13 auto sum(const std::vector<std::vector<T>>& input_matrix) -> std::enable_if_t<axis == 0,
        std::vector<T>>
14 {
15     // creating canvas
16     auto canvas {std::vector<T>(input_matrix[0].size(), 0)};
17
18     // filling up the canvas
19     for(auto row = 0; row < input_matrix.size(); ++row)
20         std::transform(input_matrix[row].begin(), input_matrix[row].end(),
21                         canvas.begin(),
22                         canvas.begin(),
23                         [](auto& argx, auto& argy){return argx + argy;});
24
25     // returning
26     return std::move(canvas);
27 }
28 /*=====
29 -----*/
30
31 template <std::size_t axis, typename T>
32 auto sum(const std::vector<std::vector<T>>& input_matrix) -> std::enable_if_t<axis == 1,
        std::vector<std::vector<T>>>
33 {
34     // creating canvas
35     auto canvas {std::vector<std::vector<T>>(input_matrix.size(),
36                                             std::vector<T>(1, 0.00))};
37
38     // filling up the canvas
39     for(auto row = 0; row < input_matrix.size(); ++row)
40         canvas[row][0] = std::accumulate(input_matrix[row].begin(),
41                                           input_matrix[row].end(),
42                                           static_cast<T>(0));
43
44     // returning
45     return std::move(canvas);
46 }
47 /*=====
48 -----*/
49
50 template <std::size_t axis, typename T>
51 auto sum(const std::vector<T>& input_vector_A,
52         const std::vector<T>& input_vector_B) -> std::enable_if_t<axis == 0,
        std::vector<T>>
53 {
54     // setup
55     const auto& num_cols_A {input_vector_A.size()};
56     const auto& num_cols_B {input_vector_B.size()};

```

```

57
58 // throwing errors
59 if (num_cols_A != num_cols_B) {std::cerr << "sum: size disparity\n";}
60
61 // creating canvas
62 auto canvas {input_vector_A};
63
64 // summing up
65 std::transform(input_vector_B.begin(), input_vector_B.end(),
66               canvas.begin(),
67               canvas.begin(),
68               std::plus<T>());
69
70 // returning
71 return std::move(canvas);
72 }

```

A.29 Tangent

```

1 #pragma once
2 namespace svr {
3     /*=====
4     y = tan-inverse(input_vector_A/input_vector_B)
5     -----*/
6     template <typename T>
7     auto atan2(const std::vector<T> input_vector_A,
8               const std::vector<T> input_vector_B)
9     {
10         // throw error
11         if (input_vector_A.size() != input_vector_B.size())
12             std::cerr << "atan2: size disparity\n";
13
14         // create canvas
15         auto canvas {std::vector<T>(input_vector_A.size(), 0)};
16
17         // performing element-wise atan2 calculation
18         std::transform(input_vector_A.begin(), input_vector_A.end(),
19                       input_vector_B.begin(),
20                       canvas.begin(),
21                       [](const auto& arg_a,
22                         const auto& arg_b){
23
24                             return std::atan2(arg_a, arg_b);
25                         });
26
27         // moving things back
28         return std::move(canvas);
29     }
30     /*=====
31     y = tan-inverse(a/b)
32     -----*/
33     template <typename T>
34     auto atan2(T scalar_A,
35               T scalar_B)
36     {
37         return std::atan2(scalar_A, scalar_B);

```

```

38     }
39 }

```

A.30 Tiling Operations

```

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

```

```

52     for(std::size_t row = 0; row < num_rows; ++row){
53         for(std::size_t col = 0; col < num_cols; ++col){
54             source_row = row % input_matrix.size();
55             source_col = col % input_matrix[0].size();
56             canvas[row][col] = input_matrix[source_row][source_col];
57         }
58     }
59
60     // returning
61     return std::move(canvas);
62 }
63 }

```

A.31 Transpose

```

1  #pragma once
2  /*=====
3  -----*/
4  template <typename T>
5  auto transpose(const std::vector<T>& input_vector){
6
7      // creating canvas
8      auto canvas {std::vector<std::vector<T>>{
9          input_vector.size(),
10         std::vector<T>(1)
11     }};
12
13     // filling canvas
14     for(auto i = 0; i < input_vector.size(); ++i){
15         canvas[i][0] = input_vector[i];
16     }
17
18     // moving it back
19     return std::move(canvas);
20 }

```

A.32 Masking

```

1  #pragma once
2  namespace svr {
3      /*=====
4      y = input_vector[mask == 1]
5      -----*/
6      template <typename T,
7              typename = std::enable_if_t< std::is_arithmetic_v<T>          ||
8              std::is_same_v<T, std::complex<double>> ||
9              std::is_same_v<T, std::complex<float>>
10              >
11              >
12      auto mask(const std::vector<T>& input_vector,
13              const std::vector<bool>& mask_vector)
14      {
15          // checking dimensionality issues
16          if (input_vector.size() != mask_vector.size())

```

```

17         std::cerr << "mask(vector, mask): incompatible size \n";
18
19     // creating canas
20     auto num_trues {std::count(mask_vector.begin(),
21                               mask_vector.end(),
22                               true)};
23     auto canvas {std::vector<T>(num_trues)};
24
25     // copying values
26     auto destination_index {0};
27     for(auto i = 0; i < input_vector.size(); ++i)
28         if (mask_vector[i] == true)
29             canvas[destination_index++] = input_vector[i];
30
31     // returning output
32     return std::move(canvas);
33 }
34 /*=====
35 -----*/
36 template <typename T>
37 auto mask(const std::vector<std::vector<T>>& input_matrix,
38           const std::vector<bool> mask_vector)
39 {
40     // fetching dimensions
41     const auto& num_rows_matrix {input_matrix.size()};
42     const auto& num_cols_matrix {input_matrix[0].size()};
43     const auto& num_cols_vector {mask_vector.size()};
44
45     // error-checking
46     if (num_cols_matrix != num_cols_vector)
47         std::cerr << "mask(matrix, bool-vector): size disparity";
48
49     // building canvas
50     auto num_trues {std::count(mask_vector.begin(),
51                               mask_vector.end(),
52                               true)};
53     auto canvas {std::vector<std::vector<T>>(
54         num_rows_matrix,
55         std::vector<T>(num_cols_vector, 0)
56     )};
57
58     // writing values
59     #pragma omp parallel for
60     for(auto row = 0; row < num_rows_matrix; ++row){
61         auto destination_index {0};
62         for(auto col = 0; col < num_cols_vector; ++col)
63             if(mask_vector[col] == true)
64                 canvas[row][destination_index++] = input_matrix[row][col];
65     }
66
67     // returning
68     return std::move(canvas);
69 }
70 /*=====
71 Fetch Indices corresponding to mask true's
72 -----*/
73 auto mask_indices(const std::vector<bool>& mask_vector)
74 {
75     // creating canvas

```

```

76     auto    num_trues {std::count(mask_vector.begin(), mask_vector.end(),
77                                   true)};
78     auto    canvas    {std::vector<std::size_t>(num_trues)};
79
80     // building canvas
81     auto    destination_index {0};
82     for(auto i = 0; i < mask_vector.size(); ++i)
83         if (mask_vector[i] == true)
84             canvas[destination_index++] = i;
85
86     // returning
87     return std::move(canvas);
88 }
89 }

```

A.33 Resetting Containers

```

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

```

A.34 Element-wise squaring

```

1  #pragma once
2  namespace svr {
3      /*=====
4      Element-wise squaring vector
5      -----*/
6      template <typename T,
7                typename = std::enable_if_t<std::is_arithmetic_v<T>>
8                >
9      auto square(const std::vector<T>& input_vector)
10     {
11         // creating canvas
12         auto canvas {std::vector<T>(input_vector.size())};
13
14         // performing calculations
15         std::transform(input_vector.begin(), input_vector.end(),
16                        canvas.begin(),
17                        [](const auto& argx){
18                            return argx * argx;
19                        });
20     }
21 }

```

```

19         });
20
21         // moving it back
22         return std::move(canvas);
23     }
24     /*=====
25     Element-wise squaring vector (in-place)
26     -----*/
27     template <typename T,
28             typename = std::enable_if_t<std::is_arithmetic_v<T>>
29             >
30     void square_inplace(std::vector<T>& input_vector)
31     {
32         // performing operations
33         std::transform(input_vector.begin(), input_vector.end(),
34             input_vector.begin(),
35             [](auto& argx){
36                 return argx * argx;
37             });
38     }
39     /*=====
40     Element-wise squaring a matrix
41     -----*/
42     template <typename T>
43     auto square(const std::vector<std::vector<T>>& input_matrix)
44     {
45         // fetching dimensions
46         const auto& num_rows {input_matrix.size()};
47         const auto& num_cols {input_matrix[0].size()};
48
49         // creating canvas
50         auto canvas {std::vector<std::vector<T>>(
51             num_rows,
52             std::vector<T>(num_cols, 0)
53         )};
54
55         // going through each row
56         #pragma omp parallel for
57         for(auto row = 0; row < num_rows; ++row)
58             std::transform(input_matrix[row].begin(), input_matrix[row].end(),
59                 canvas[row].begin(),
60                 [](const auto& argx){
61                     return argx * argx;
62                 });
63
64         // returning
65         return std::move(canvas);
66     }
67     /*=====
68     Squaring for scalars
69     -----*/
70     template <typename T>
71     auto square(const T& scalar) {return scalar * scalar;}
72 }

```

A.35 Flooring

```

1 namespace svr {
2     /*=====
3     element-wise flooring of a vector-contents
4     -----*/
5     template <typename T>
6     auto floor(const std::vector<T>& input_vector)
7     {
8         // creating canvas
9         auto canvas {std::vector<T>(input_vector.size())};
10
11        // filling the canvas
12        std::transform(input_vector.begin(), input_vector.end(),
13                       canvas.begin(),
14                       [](const auto& argx){
15                           return std::floor(argx);
16                       });
17
18        // returning
19        return std::move(canvas);
20    }
21    /*=====
22    element-wise flooring of a vector-contents (in-place)
23    -----*/
24    template <typename T>
25    auto floor_inplace(std::vector<T>& input_vector)
26    {
27        // rewriting the contents
28        std::transform(input_vector.begin(), input_vector.end(),
29                       input_vector.begin(),
30                       [](auto& argx){
31                           return std::floor(argx);
32                       });
33    }
34    /*=====
35    element-wise flooring of matrix-contents
36    -----*/
37    template <typename T>
38    auto floor(const std::vector<std::vector<T>>& input_matrix)
39    {
40        // fetching dimensions
41        const auto& num_rows_matrix {input_matrix.size()};
42        const auto& num_cols_matrix {input_matrix[0].size()};
43
44        // creating canvas
45        auto canvas {std::vector<std::vector<T>>(
46            num_rows_matrix,
47            std::vector<T>(num_cols_matrix)
48        )};
49
50        // writing contents
51        for(auto row = 0; row < num_rows_matrix; ++row)
52            std::transform(input_matrix[row].begin(), input_matrix[row].end(),
53                           canvas[row].begin(),
54                           [](const auto& argx){
55                               return std::floor(argx);
56                           });
57
58        // returning contents
59        return std::move(canvas);

```

```

60 }
61
62 /*=====
63 element-wise flooring of matrix-contents (in-place)
64 -----*/
65 template <typename T>
66 auto floor_inplace(std::vector<std::vector<T>>& input_matrix)
67 {
68     // performing operations
69     for(auto row = 0; row < input_matrix.size(); ++row)
70         std::transform(input_matrix[row].begin(), input_matrix[row].end(),
71                        input_matrix[row].begin(),
72                        [](auto& argx){
73                            return std::floor(argx);
74                        });
75 }
76 }

```

A.36 Squeeze

```

1 namespace svr {
2     template <typename T>
3     auto squeeze(const std::vector<std::vector<T>>& input_matrix)
4     {
5         // fetching dimensions
6         const auto& num_rows_matrix {input_matrix.size()};
7         const auto& num_cols_matrix {input_matrix[0].size()};
8
9         // check if any dimension is 1
10        if (num_rows_matrix == 0 || num_cols_matrix == 0)
11            std::cerr << "at least one dimension should be 1";
12
13        auto final_length {std::max(num_rows_matrix, num_cols_matrix)};
14
15        // creating canvas
16        auto canvas {std::vector<T>(final_length)};
17
18        // building canvas
19        if (num_rows_matrix == 1)
20        {
21            // filling canvas
22            std::copy(input_matrix[0].begin(), input_matrix[0].end(),
23                    canvas.begin());
24        }
25        else if (num_cols_matrix == 1)
26        {
27            // filling canvas
28            std::transform(input_matrix.begin(), input_matrix.end(),
29                           canvas.begin(),
30                           [](const auto& argx){
31                               return argx[0];
32                           });
33        }
34
35        // returning
36        return std::move(canvas);

```

```

37     }
38 }

```

A.37 Tensor Initializations

```

1  namespace svr {
2      /*=====
3      -----*/
4      template <typename T>
5      auto zeros(const std::array<std::size_t, 2> input_dimensions)
6      {
7          // create canvas
8          auto canvas {std::vector<std::vector<T>>(
9              input_dimensions[0],
10             std::vector<T>(input_dimensions[1], 0)
11         )};
12
13         // returning
14         return std::move(canvas);
15     }
16 }

```

A.38 Real part

```

1  #pragma once
2  namespace svr {
3
4      /*=====
5      For type-deductions
6      -----*/
7      template <typename T>
8      struct real_result_type;
9
10     template <> struct real_result_type<std::complex<double>>{
11         using type = double;
12     };
13     template <> struct real_result_type<std::complex<float>>{
14         using type = float;
15     };
16     template <> struct real_result_type<double> {
17         using type = double;
18     };
19     template <> struct real_result_type<float>{
20         using type = float;
21     };
22
23     template <typename T>
24     using real_result_t = typename real_result_type<T>::type;
25
26     /*=====
27     Element-wise real() of a vector
28     -----*/
29     template <typename T>
30     auto real(const std::vector<T>& input_vector)

```

```

31 {
32     // figure out base-type
33     using TCanvas = real_result_t<T>;
34
35     // creating canvas
36     auto canvas = {std::vector<TCanvas>(
37         input_vector.size()
38     )};
39
40     // storing values
41     std::transform(input_vector.begin(), input_vector.end(),
42                   canvas.begin(),
43                   [](const auto& argx){
44                       return std::real(argx);
45                   });
46
47     // returning
48     return std::move(canvas);
49 }
50 // /*=====
51 // Element-wise real() of a matrix
52 // -----*/
53 // template <typename T>
54 // auto real(const std::vector<std::vector<T>>& input_matrix)
55 // {
56 //     // fetching dimensions
57 //     const auto& num_rows_matrix = {input_matrix.size()};
58 //     const auto& num_cols_matrix = {input_matrix[0].size()};
59
60 //     // creating canvas
61 //     auto canvas = {std::vector<std::vector<T>>(
62 //         num_rows_matrix,
63 //         std::vector<T>(num_cols_matrix)
64 //     )};
65
66 //     // storing the values
67 //     for(auto row = 0; row < num_rows_matrix; ++row)
68 //         std::transform(input_matrix[row].begin(), input_matrix[row].end(),
69 //                         canvas[row].begin(),
70 //                         [](const auto& argx){
71 //                             return std::real(argx);
72 //                         });
73
74 //     // returning
75 //     return std::move(canvas);
76 // }
77 }

```

A.39 Imaginary part

```

1 #pragma once
2 namespace svr {
3
4     /*=====
5     For type-deductions
6     -----*/

```

```

7  template <typename T>
8  struct imag_result_type;
9
10 template <> struct imag_result_type<std::complex<double>>{
11     using type = double;
12 };
13 template <> struct imag_result_type<std::complex<float>>{
14     using type = float;
15 };
16 template <> struct imag_result_type<double> {
17     using type = double;
18 };
19 template <> struct imag_result_type<float>{
20     using type = float;
21 };
22
23 template <typename T>
24 using imag_result_t = typename imag_result_type<T>::type;
25
26 /*=====
27 -----*/
28 template <typename T>
29 auto imag(const std::vector<T>& input_vector)
30 {
31     // figure out base-type
32     using TCanvas = imag_result_t<T>;
33
34     // creating canvas
35     auto canvas {std::vector<TCanvas>(
36         input_vector.size()
37     )};
38
39     // storing values
40     std::transform(input_vector.begin(), input_vector.end(),
41                   canvas.begin(),
42                   [](const auto& argx){
43                       return std::imag(argx);
44                   });
45
46     // returning
47     return std::move(canvas);
48 }
49 }

```

Appendix B

Software Tools

B.1 CSV File-Writes

```
1 #pragma once
2 /*=====
3 writing the contents of a vector a csv-file
4 -----*/
5 template <typename T>
6 void fWriteVector(const vector<T>&          inputvector,
7                  const string&             filename){
8
9     // opening a file
10    std::ofstream fileobj(filename);
11    if (!fileobj) {return;}
12
13    // writing the real parts in the first column and the imaginary parts int he second
14    // column
15    if constexpr(std::is_same_v<T, std::complex<double>> ||
16                  std::is_same_v<T, std::complex<float>> ||
17                  std::is_same_v<T, std::complex<long double>>){
18        for(int i = 0; i<inputvector.size(); ++i){
19            // adding entry
20            fileobj << inputvector[i].real() << "+" << inputvector[i].imag() << "i";
21
22            // adding delimiter
23            if(i!=inputvector.size()-1) {fileobj << ",";}
24            else {fileobj << "\n";}
25        }
26    }
27    else{
28        for(int i = 0; i<inputvector.size(); ++i){
29            fileobj << inputvector[i];
30            if(i!=inputvector.size()-1) {fileobj << ",";}
31            else {fileobj << "\n";}
32        }
33    }
34
35    // return
36    return;
37 }
38 /*=====
```

```

38  writing the contents of a matrix to a csv-file
39  -----*/
40  template <typename T>
41  auto fWriteMatrix(const std::vector<std::vector<T>> inputMatrix,
42                  const string filename){
43
44      // opening a file
45      std::ofstream fileobj(filename);
46
47      // writing
48      if (fileobj){
49          for(int i = 0; i<inputMatrix.size(); ++i){
50              for(int j = 0; j<inputMatrix[0].size(); ++j){
51                  fileobj << inputMatrix[i][j];
52                  if (j!=inputMatrix[0].size()-1) {fileobj << ",";}
53                  else {fileobj << "\n";}
54              }
55          }
56      }
57      else{
58          cout << format("File-write to {} failed\n", filename);
59      }
60  }
61
62  /*=====
63  writing complex-matrix to a csv-file
64  -----*/
65  template <>
66  auto fWriteMatrix(const std::vector<std::vector<std::complex<double>>> inputMatrix,
67                  const string filename){
68
69      // opening a file
70      std::ofstream fileobj(filename);
71
72      // writing
73      if (fileobj){
74          for(int i = 0; i<inputMatrix.size(); ++i){
75              for(int j = 0; j<inputMatrix[0].size(); ++j){
76                  fileobj << inputMatrix[i][j].real() << "+" << inputMatrix[i][j].imag() <<
77                      "i";
78                  if (j!=inputMatrix[0].size()-1) {fileobj << ",";}
79                  else {fileobj << "\n";}
80              }
81          }
82      }
83      else{
84          cout << format("File-write to {} failed\n", filename);
85      }
86  }

```

B.2 Thread-Pool

```

1  #pragma once
2  namespace svr {
3      class ThreadPool {
4      public:

```

```

5      // Members
6      boost::asio::thread_pool      thread_pool;      // the pool
7      std::vector<std::future<void>> future_vector;    // futures to wait on
8
9      // Special-Members
10     ThreadPool(std::size_t num_threads) : 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 }

```

B.3 FFTPlanClass

```

1  #pragma once
2
3  namespace svr    {
4
5      template <typename sourceType,
6                typename destinationType,
7                typename = std::enable_if_t<std::is_same_v<sourceType, double> &&
8                                     std::is_same_v<destinationType,
9                                     std::complex<double>>
10                                     >
11                                     >
12  class FFTPlanClass
13  {
14  public:
15
16      // Members
17      std::size_t    nfft_      {std::numeric_limits<std::size_t>::max()};
18      fftw_complex*  in_        {nullptr};
19      fftw_complex*  out_       {nullptr};
20      fftw_plan      plan_      {nullptr};
21
22      /*=====
23      Destructor
24      -----*/
25      ~FFTPlanClass()
26      {
27          if (plan_ != nullptr) {fftw_destroy_plan( plan_);}
28          fftw_free( in_);
29          fftw_free( out_);
30      }
31      /*=====
32      Default Constructor
33      -----*/
34      FFTPlanClass() = default;
35      /*=====
36      Constructor
37      -----*/
38      FFTPlanClass(const std::size_t nfft)
39      {
40          // allocating nfft
41          this->nfft_ = nfft;
42
43          // allocating input-region
44          in_ = reinterpret_cast<fftw_complex*>(
45              fftw_malloc(nfft_ * sizeof(fftw_complex))
46          );
47          out_ = reinterpret_cast<fftw_complex*>(
48              fftw_malloc(nfft_ * sizeof(fftw_complex))
49          );
50          if(!in_ || !out_) {throw std::runtime_error("FILE: FFTPlanClass.hpp |
51              CLASS: FFTPlanClass | REPORT: in-out allocation failed");}
52
53          // creating plan
54          plan_ = fftw_plan_dft_1d(
55              static_cast<int>(nfft_),
56              in_,
57              out_,
58              FFTW_FORWARD,
59              FFTW_MEASURE
60          );
61      }
62  };
63  }

```

```

58         );
59         if(!plan_)           {throw std::runtime_error("FILE: FFTPlanClass.hpp |
                                CLASS: FFTPlanClass | REPORT: plan-creation failed");}
60     }
61     /*=====
62     Copy Constructor
63     -----*/
64     FFTPlanClass(const FFTPlanClass& other)
65     {
66         // copying nfft
67         nfft_ = other.nfft_;
68         cout << format("\t\t FFTPlanClass(const FFTPlanClass& other) | nfft_ =
                                {}\n", nfft_);
69
70         // allocating input-region
71         in_ = reinterpret_cast<fftw_complex*>(
72             fftw_malloc(nfft_ * sizeof(fftw_complex))
73         );
74         out_ = reinterpret_cast<fftw_complex*>(
75             fftw_malloc(nfft_ * sizeof(fftw_complex))
76         );
77         if(!in_ || !out_) {throw std::runtime_error("FILE: FFTPlanClass.hpp |
                                CLASS: FFTPlanClass | REPORT: in-out allocation failed");}
78
79         // copying input-region and output-region
80         std::memcpy(in_, other.in_, nfft_ * sizeof(fftw_complex));
81         std::memcpy(out_, other.out_, nfft_ * sizeof(fftw_complex));
82
83         // creating plan
84         plan_ = fftw_plan_dft_1d(
85             static_cast<int>(nfft_),
86             in_,
87             out_,
88             FFTW_FORWARD,
89             FFTW_MEASURE
90         );
91         if(!plan_)           {throw std::runtime_error("FILE: FFTPlanClass.hpp |
                                CLASS: FFTPlanClass | REPORT: plan-creation failed");}
92     }
93     /*=====
94     Copy Assignment
95     -----*/
96     FFTPlanClass& operator=(const FFTPlanClass& other)
97     {
98         // handling self-assignment
99         if (this == &other) {return *this;}
100
101         // cleaning-up existing resources
102         fftw_destroy_plan( plan_);
103         fftw_free( in_);
104         fftw_free( out_);
105
106         // allocating input-region and output-region
107         nfft_ = other.nfft_;
108         in_ = reinterpret_cast<fftw_complex*>(
109             fftw_malloc(nfft_ * sizeof(fftw_complex))
110         );
111         out_ = reinterpret_cast<fftw_complex*>(
112             fftw_malloc(nfft_ * sizeof(fftw_complex))

```

```

113     );
114     if(!in_ || !out_) {throw std::runtime_error("FILE: FFTPlanClass.hpp |
115         CLASS: FFTPlanClass | FUNCTION: Copy-Assignment | REPORT: in-out
116         allocation failed");}
117
118     // copying contents
119     cout << format("\t\t FFTPlanClass& operator=(const FFTPlanClass& other) |
120         nfft_ = {}\n", nfft_);
121     std::memcpy(in_, other.in_, nfft_ * sizeof(fftw_complex));
122     std::memcpy(out_, other.out_, nfft_ * sizeof(fftw_complex));
123
124     // creating engine
125     plan_ = fftw_plan_dft_id(
126         static_cast<int>(nfft_),
127         in_,
128         out_,
129         FFTW_FORWARD,
130         FFTW_MEASURE
131     );
132     if(!plan_) {throw std::runtime_error("FILE: FFTPlanClass.hpp | CLASS:
133         FFTPlanClass | FUNCTION: Copy-Assignment | REPORT: plan-creation
134         failed");}
135
136     // returning
137     return *this;
138 }
139
140 /*=====
141 Move Constructor
142 -----*/
143 FFTPlanClass(FFTPlanClass&& other)
144 :   nfft_(      other.nfft_),
145   in_(          other.in_),
146   out_(         other.out_),
147   plan_(        other.plan_)
148 {
149     // resetting the others
150     other.nfft_   = 0;
151     other.in_     = nullptr;
152     other.out_    = nullptr;
153     other.plan_   = nullptr;
154 }
155
156 /*=====
157 Move Assignment
158 -----*/
159 FFTPlanClass& operator=(FFTPlanClass&& other)
160 {
161     // self-assignment check
162     if (this == &other) {return *this;}
163
164     // cleaning up existing resources
165     fftw_destroy_plan(   plan_);
166     fftw_free(           in_);
167     fftw_free(           out_);
168
169     // Copying-values and changing pointers
170     nfft_ = other.nfft_;
171     cout << format("\t\t FFTPlanClass& operator=(FFTPlanClass&& other) | nfft_
172         = {}\n", nfft_);
173     in_ = other.in_;

```

```

166         out_                = other.out_;
167         plan_                = other.plan_;
168
169         // resetting source-members
170         other.nfft_          = 0;
171         other.in_            = nullptr;
172         other.out_           = nullptr;
173         other.plan_          = nullptr;
174
175         // returning
176         return *this;
177     }
178     /*=====
179     Running fft
180     -----*/
181     std::vector<destinationType>
182     fft(const std::vector<sourceType>& input_vector)
183     {
184         // throwing an error
185         if (input_vector.size() > nfft_){
186             cout << format("input_vector.size() = {}, nfft_ = {}\n",
187                           input_vector.size(),
188                           nfft_);
189             throw std::runtime_error("FILE: FFTPlanClass.hpp | CLASS: FFTPlanClass
190                                     | FUNCTION: fft() | REPORT: input-vector size is greater than
191                                     NFFT");
192         }
193
194         // copying inputs
195         for(std::size_t index = 0; index < input_vector.size(); ++index)
196         {
197             if constexpr(
198                 std::is_same_v< sourceType, double >
199             ){
200                 in_[index][0] = input_vector[index];
201                 in_[index][1] = 0;
202             }
203             else if constexpr(
204                 std::is_same_v< sourceType, std::complex<double> >
205             ){
206                 in_[index][0] = input_vector[index].real();
207                 in_[index][1] = input_vector[index].imag();
208             }
209         }
210
211         // executing fft
212         fftw_execute(plan_);
213
214         // copying results to output-vector
215         std::vector<destinationType> output_vector(nfft_);
216         for(std::size_t index = 0; index < nfft_; ++index){
217             if constexpr(
218                 std::is_same_v< destinationType, std::complex<double> >
219             ){
220                 output_vector[index] = std::complex<double>(
221                     out_[index][0],
222                     out_[index][1]

```

```

223     }
224     else if constexpr(
225         std::is_same_v< destinationType, double >
226     ){
227         output_vector[index] = std::sqrt(
228             std::pow(out_[index][0], 2) + \
229             std::pow(out_[index][1], 2)
230         );
231     }
232 }
233
234 // returning output
235 return std::move(output_vector);
236 }
237 /*=====
238 Running fft - balanced
239 -----*/
240 std::vector<destinationType>
241 fft_l2_conserved(const std::vector<sourceType>& input_vector)
242 {
243     // throwing an error
244     if (input_vector.size() > nfft_)
245         throw std::runtime_error("FILE: FFTPlanClass.hpp | CLASS: FFTPlanClass
246                                     | FUNCTION: fft() | REPORT: input-vector size is greater than
247                                     NFFT");
248
249     // copying inputs
250     for(std::size_t index = 0; index < input_vector.size(); ++index)
251     {
252         if constexpr(
253             std::is_same_v< sourceType, double >
254         ){
255             in_[index][0] = input_vector[index];
256             in_[index][1] = 0;
257         }
258         else if constexpr(
259             std::is_same_v< sourceType, std::complex<double> >
260         ){
261             in_[index][0] = input_vector[index].real();
262             in_[index][1] = input_vector[index].imag();
263         }
264     }
265
266     // executing fft
267     fftw_execute(plan_);
268
269     // copying results to output-vector
270     std::vector<destinationType> output_vector(nfft_);
271     for(std::size_t index = 0; index < nfft_; ++index){
272         if constexpr(
273             std::is_same_v< destinationType, std::complex<double> >
274         ){
275             output_vector[index] = std::complex<double>(
276                 out_[index][0] * (1.00 / std::sqrt(nfft_)),
277                 out_[index][1] * (1.00 / std::sqrt(nfft_))
278             );
279         }
280         else if constexpr(
281             std::is_same_v< destinationType, double >

```

```

280         ){
281             output_vector[index] = std::sqrt(
282                 std::pow(out_[index][0], 2) + \
283                 std::pow(out_[index][1], 2)
284             );
285         }
286     }
287
288     // returning output
289     return std::move(output_vector);
290 }
291 };
292 }

```

B.4 IFFTPlanClass

```

1  #pragma once
2  namespace svr {
3      template <typename sourceType,
4               typename destinationType,
5               typename = std::enable_if_t<std::is_same_v<sourceType,
6               std::complex<double>> &&
7               std::is_same_v<destinationType, double>
8               >
9      class IFFTPlanClass
10     {
11     public:
12         std::size_t      nfft_;
13         fftw_complex*     in_;
14         fftw_complex*     out_;
15         fftw_plan         plan_;
16
17         /*=====
18         Destructor
19         -----*/
20         ~IFFTPlanClass()
21         {
22             fftw_destroy_plan(plan_);
23             fftw_free(in_);
24             fftw_free(out_);
25         }
26         /*=====
27         Constructor
28         -----*/
29         IFFTPlanClass(const std::size_t nfft): nfft_(nfft)
30         {
31             // allocating space
32             in_ = reinterpret_cast<fftw_complex*>(fftw_malloc(nfft_ *
33                 sizeof(fftw_complex)));
34             out_ = reinterpret_cast<fftw_complex*>(fftw_malloc(nfft_ *
35                 sizeof(fftw_complex)));
36             if(!in_ || !out_) {throw std::runtime_error("in_, out_ creation
37                 failed");}
38
39             // creating plan

```

```

37     plan_ = fftw_plan_dft_1d(
38         static_cast<int>(nfft_),
39         in_,
40         out_,
41         FFTW_BACKWARD,
42         FFTW_MEASURE
43     );
44     if(!plan_) {throw std::runtime_error("File: FFTPlanClass.hpp |
45         Class: IFFTPlanClass | report: plan-creation failed");}
46 }
47 /*=====
48 Copy Constructor
49 -----*/
50 IFFTPlanClass(const IFFTPlanClass& other)
51 {
52     // allocating space
53     nfft_ = other.nfft_;
54     in_ = reinterpret_cast<fftw_complex*>(fftw_malloc(nfft_ *
55         sizeof(fftw_complex)));
56     out_ = reinterpret_cast<fftw_complex*>(fftw_malloc(nfft_ *
57         sizeof(fftw_complex)));
58     if (!in_ || !out_) {throw std::runtime_error("FILE: FFTPlanClass.hpp |
59         Class: IFFTPlanClass | Function: Copy-Constructor | Report: in-out
60         plan creation failed");}
61
62     // copying contents
63     std::memcpy(in_, other.in_, nfft_ * sizeof(fftw_complex));
64     std::memcpy(out_, other.out_, nfft_ * sizeof(fftw_complex));
65
66     // creating a new plan since its more of an engine
67     plan_ = fftw_plan_dft_1d(
68         static_cast<int>(nfft_),
69         in_,
70         out_,
71         FFTW_BACKWARD,
72         FFTW_MEASURE
73     );
74     if(!plan_) {throw std::runtime_error("FILE: FFTPlanClass.hpp | Class:
75         IFFTPlanClass | Function: Copy-Constructor | Report: plan-creation
76         failed");}
77 }
78 /*=====
79 Copy Assignment
80 -----*/
81 IFFTPlanClass& operator=(const IFFTPlanClass& other)
82 {
83     // handling self-assignment
84     if(this != &other) {return *this;}
85
86     // cleaning up existing resources
87     fftw_destroy_plan(plan_);
88     fftw_free(in_);
89     fftw_free(out_);
90
91     // allocating space
92     nfft_ = other.nfft_;
93     in_ = reinterpret_cast<fftw_complex*>(fftw_malloc(nfft_ *
94         sizeof(fftw_complex)));

```

```

88     out_ = reinterpret_cast<fftw_complex*>(fftw_malloc(nfft_ *
89         sizeof(fftw_complex)));
90     if (!in_ || !out_) {throw std::runtime_error("FILE: FFTPlanClass.hpp |
91         Class: IFFTPlanClass | Function: Copy-Constructor | Report: in-out
92         plan creation failed");}
93
94     // copying contents
95     std::memcpy(in_, other.in_, nfft_ * sizeof(fftw_complex));
96     std::memcpy(out_, other.out_, nfft_ * sizeof(fftw_complex));
97
98     // creating a new plan since its more of an engine
99     plan_ = fftw_plan_dft_id(
100         static_cast<int>(nfft_),
101         in_,
102         out_,
103         FFTW_BACKWARD,
104         FFTW_MEASURE
105     );
106     if(!plan_) {throw std::runtime_error("FILE: FFTPlanClass.hpp | Class:
107         IFFTPlanClass | Function: Copy-Constructor | Report: plan-creation
108         failed");}
109
110     // returning
111     return *this;
112 }
113
114 /*=====
115 Move Constructor
116 -----*/
117 IFFTPlanClass(IFFTPlanClass&& other) noexcept
118 :   nfft_( other.nfft_),
119     in_(   other.in_),
120     out_(  other.out_),
121     plan_( other.plan_)
122 {
123     // resetting the source object
124     other.nfft_ = 0;
125     other.in_   = nullptr;
126     other.out_  = nullptr;
127     other.plan_ = nullptr;
128 }
129
130 /*=====
131 Move-Assignment
132 -----*/
133 IFFTPlanClass& operator=(IFFTPlanClass&& other) noexcept
134 {
135     // self-assignment check
136     if(this != &other) {return *this;}
137
138     // cleaning up existing
139     fftw_destroy_plan( plan_);
140     fftw_free(         in_);
141     fftw_free(         out_);
142
143     // Copying values and changing pointers
144     nfft_ = other.nfft_;
145     in_   = other.in_;
146     out_  = other.out_;
147     plan_ = other.plan_;

```



```

142
143     // resetting the source-object
144     other.nfft_ = 0;
145     other.in_   = nullptr;
146     other.out_  = nullptr;
147     other.plan_ = nullptr;
148
149     // returning
150     return *this;
151 }
152 /*=====
153 Running
154 -----*/
155 std::vector<destinationType>
156 ifft(const std::vector<sourceType>& input_vector)
157 {
158     // throwing an error
159     if (input_vector.size() > nfft_)
160         throw std::runtime_error("File: FFTPlanClass | Class: IFFTPlanClass |
161                                 Function: ifft() | Report: size of vector > nfft ");
162
163     // copy input into fftw buffer
164     for(std::size_t index = 0; index < nfft_; ++index)
165     {
166         if constexpr(
167             std::is_same_v< sourceType, std::complex<double> >
168         ){
169             in_[index][0] = input_vector[index].real();
170             in_[index][1] = input_vector[index].imag();
171         }
172         else if constexpr(
173             std::is_same_v< sourceType, double >
174         ){
175             in_[index][0] = input_vector[index];
176             in_[index][1] = 0;
177         }
178     }
179
180     // execute ifft
181     fftw_execute(plan_);
182
183     // normalize output
184     std::vector<destinationType> output_vector(nfft_);
185     for(std::size_t index = 0; index < nfft_; ++index){
186         if constexpr(
187             std::is_same_v< destinationType, double >
188         ){
189             output_vector[index] = out_[index][0]/nfft_;
190         }
191         else if constexpr(
192             std::is_same_v< destinationType, std::complex<double> >
193         ){
194             output_vector[index][0] = std::complex<double>(
195                 out_[index][0]/nfft_,
196                 out_[index][1]/nfft_
197             );
198         }
199     }

```

```

200     // returning
201     return std::move(output_vector);
202 }
203 /*=====
204 Running - proper bases change
205 -----*/
206 std::vector<destinationType>
207 ifft_l2_conserved(const std::vector<sourceType>& input_vector)
208 {
209     // throwing an error
210     if (input_vector.size() > nfft_)
211         throw std::runtime_error("File: FFTPlanClass | Class: IFFTPlanClass |
212                                     Function: ifft() | Report: size of vector > nfft ");
213
214     // copy input into fftw buffer
215     for(std::size_t index = 0; index < nfft_; ++index)
216     {
217         if constexpr(
218             std::is_same_v< sourceType, std::complex<double> >
219         ){
220             in_[index][0] = input_vector[index].real();
221             in_[index][1] = input_vector[index].imag();
222         }
223         else if constexpr(
224             std::is_same_v< sourceType, double >
225         ){
226             in_[index][0] = input_vector[index];
227             in_[index][1] = 0;
228         }
229     }
230
231     // execute ifft
232     fftw_execute(plan_);
233
234     // normalize output
235     std::vector<destinationType> output_vector(nfft_);
236     for(std::size_t index = 0; index < nfft_; ++index){
237         if constexpr(
238             std::is_same_v< destinationType, double >
239         ){
240             output_vector[index] = out_[index][0]/std::sqrt(nfft_);
241         }
242         else if constexpr(
243             std::is_same_v< destinationType, std::complex<double> >
244         ){
245             output_vector[index][0] = std::complex<double>(
246                 out_[index][0]/nfft_,
247                 out_[index][1]/nfft_
248             );
249         }
250     }
251
252     // returning
253     return std::move(output_vector);
254 }
255 }

```

B.5 FFT Plan Pool

```

1 namespace svr {
2     template <typename sourceType,
3               typename destinationType,
4               typename = std::enable_if_t<std::is_same_v<sourceType, double> &&
5                                     std::is_same_v<destinationType,
6                                     std::complex<double>>
7                                     >
8 class FFTPlanUniformPool
9 {
10     public:
11
12         // Members
13         struct AccessPairs
14         {
15             // members
16             svr::FFTPlanClass <sourceType, destinationType> plan;
17             std::mutex mutex;
18
19             // special members
20             AccessPairs() = default;
21             AccessPairs(const std::size_t nfft): plan(nfft) {}
22             AccessPairs(const AccessPairs& other) = default;
23             AccessPairs& operator=(const AccessPairs& other) = default;
24             AccessPairs(AccessPairs&& other) = default;
25             AccessPairs& operator=(AccessPairs&& other) = default;
26         };
27         std::vector< std::unique_ptr<AccessPairs> > engines;
28
29
30         // Special Member Function
31         FFTPlanUniformPool() = default;
32         FFTPlanUniformPool(const std::size_t num_plans,
33                             const std::size_t nfft)
34         {
35             engines.reserve(num_plans);
36             for(auto i = 0; i < num_plans; ++i)
37                 engines.emplace_back(std::make_unique<AccessPairs>(nfft));
38         }
39         FFTPlanUniformPool(const FFTPlanUniformPool& other) = delete;
40         FFTPlanUniformPool& operator=(const FFTPlanUniformPool& other) = delete;
41         FFTPlanUniformPool(FFTPlanUniformPool&& other) = default;
42         FFTPlanUniformPool& operator=(FFTPlanUniformPool&& other) = default;
43
44
45         // Members
46         AccessPairs& fetch_plan()
47         {
48             auto num_rounds {12};
49             for(auto i = 0; i < num_rounds; ++i){
50                 for(auto& engine_ptr: engines){
51                     if(engine_ptr->mutex.try_lock()){
52                         engine_ptr->mutex.unlock();
53                         return *engine_ptr;
54                     }
55                 }
56             }

```

```

57         throw std::runtime_error("FILE: FFTPlanPoolClass.hpp | CLASS:
58             FFTPlanUniformPool | FUNCTION: fetch_plan() | Report: No plans
59             available despite num-rounds rounds of searching");
60     };
61 }

```

B.6 IFFT Plan Pool

```

1  namespace svr {
2      template <typename sourceType,
3              typename destinationType,
4              typename = std::enable_if_t<std::is_same_v<sourceType, double> &&
5                  std::is_same_v<destinationType,
6                      std::complex<double>>
7                  >
8      class IFFTPlanUniformPool
9      {
10     public:
11
12         // Members
13         struct AccessPairs
14         {
15             // members
16             svr::IFFTPlanClass <sourceType, destinationType> plan;
17             std::mutex mutex;
18
19             // special members
20             AccessPairs() = default;
21             AccessPairs(const std::size_t nfft):
22                 plan(nfft) {}
23             AccessPairs(const AccessPairs& other) = default;
24             AccessPairs& operator=(const AccessPairs& other) = default;
25             AccessPairs(AccessPairs&& other) = default;
26             AccessPairs& operator=(AccessPairs&& other) = default;
27         };
28         std::vector< std::unique_ptr<AccessPairs> > engines;
29
30         // Special Member Function
31         IFFTPlanUniformPool() = default;
32         IFFTPlanUniformPool(const std::size_t num_plans,
33                             const std::size_t nfft)
34         {
35             engines.reserve(num_plans);
36             for(auto i = 0; i<num_plans; ++i)
37                 engines.emplace_back(std::make_unique<AccessPairs>(nfft));
38         }
39         IFFTPlanUniformPool(const IFFTPlanUniformPool& other) = delete;
40         IFFTPlanUniformPool& operator=(const IFFTPlanUniformPool& other) = delete;
41         IFFTPlanUniformPool(IFFTPlanUniformPool&& other) = default;
42         IFFTPlanUniformPool& operator=(IFFTPlanUniformPool&& other) = default;
43
44         // Members
45         AccessPairs& fetch_plan()
46         {

```

```
47         auto num_rounds    {12};
48         for(auto i = 0; i < num_rounds; ++i){
49             for(auto& engine_ptr: engines){
50                 if(engine_ptr->mutex.try_lock()){
51                     engine_ptr->mutex.unlock();
52                     return *engine_ptr;
53                 }
54             }
55         }
56         throw std::runtime_error("FILE: IFFTPlanPoolClass.hpp | CLASS:
           IFFTPlanUniformPool | FUNCTION: fetch_plan() | Report: No plans
           available despite num-rounds rounds of searching");
57
58     }
59 };
60 }
```
