

# Autonomous Underwater Vehicle: A Surveillance Protocol

S.V. Rajendran

October 11, 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

<b>Preface</b>	<b>i</b>
<b>I AUV Components &amp; Setup</b>	<b>1</b>
<b>1 Underwater Environment</b>	<b>2</b>
1.1 Underwater Hills . . . . .	2
1.2 Scatterer Definition . . . . .	3
1.3 Sea-Floor Setup Script . . . . .	4
<b>2 Transmitter</b>	<b>6</b>
2.1 Transmission Signal . . . . .	7
2.2 Transmitter Class Definition . . . . .	8
2.3 Transmitter Setup Scripts . . . . .	9
<b>3 Uniform Linear Array</b>	<b>13</b>
3.1 ULA Class Definition . . . . .	14
3.2 ULA Setup Scripts . . . . .	16
<b>4 Autonomous Underwater Vehicle</b>	<b>18</b>
4.1 AUV Class Definition . . . . .	19
4.2 AUV Setup Scripts . . . . .	20
<b>II Signal Simulation Pipeline</b>	<b>21</b>
<b>5 Signal Simulation</b>	<b>22</b>

### III Imaging Pipeline 24

### IV Perception & Control Pipeline 25

#### A Application Specific Tools 26

A.1	CSV File-Writes . . . . .	26
A.2	Thread-Pool . . . . .	27
A.3	FFTPlanClass . . . . .	28
A.4	IFFTPlanClass . . . . .	34
A.5	FFT Plan Pool . . . . .	39
A.6	IFFT Plan Pool . . . . .	40
A.7	FFT Plan Pool Handle . . . . .	42
A.8	IFFT Plan Pool Handle . . . . .	43

#### B General Purpose Templated Functions 45

B.1	abs . . . . .	45
B.2	Boolean Comparators . . . . .	46
B.3	Concatenate Functions . . . . .	47
B.4	Conjugate . . . . .	49
B.5	Convolution . . . . .	49
B.6	Coordinate Change . . . . .	59
B.7	Cosine . . . . .	61
B.8	Data Structures . . . . .	62
B.9	Editing Index Values . . . . .	62
B.10	Equality . . . . .	63
B.11	Exponentiate . . . . .	64
B.12	FFT . . . . .	65
B.13	Flipping Containers . . . . .	69
B.14	Indexing . . . . .	70
B.15	Linspace . . . . .	72
B.16	Max . . . . .	73
B.17	Meshgrid . . . . .	74
B.18	Minimum . . . . .	75
B.19	Norm . . . . .	75
B.20	Division . . . . .	77
B.21	Addition . . . . .	79

B.22	Multiplication (Element-wise) . . . . .	82
B.23	Subtraction . . . . .	87
B.24	Printing Containers . . . . .	89
B.25	Random Number Generation . . . . .	90
B.26	Reshape . . . . .	92
B.27	Summing with containers . . . . .	94
B.28	Tangent . . . . .	96
B.29	Tiling Operations . . . . .	97
B.30	Transpose . . . . .	98
B.31	Masking . . . . .	98
B.32	Resetting Containers . . . . .	100
B.33	Element-wise squaring . . . . .	100
B.34	Flooring . . . . .	101
B.35	Squeeze . . . . .	103
B.36	Tensor Initializations . . . . .	104
B.37	Real part . . . . .	104
B.38	Imaginary part . . . . .	105

## **Part I**

# **AUV Components & Setup**

# Chapter 1

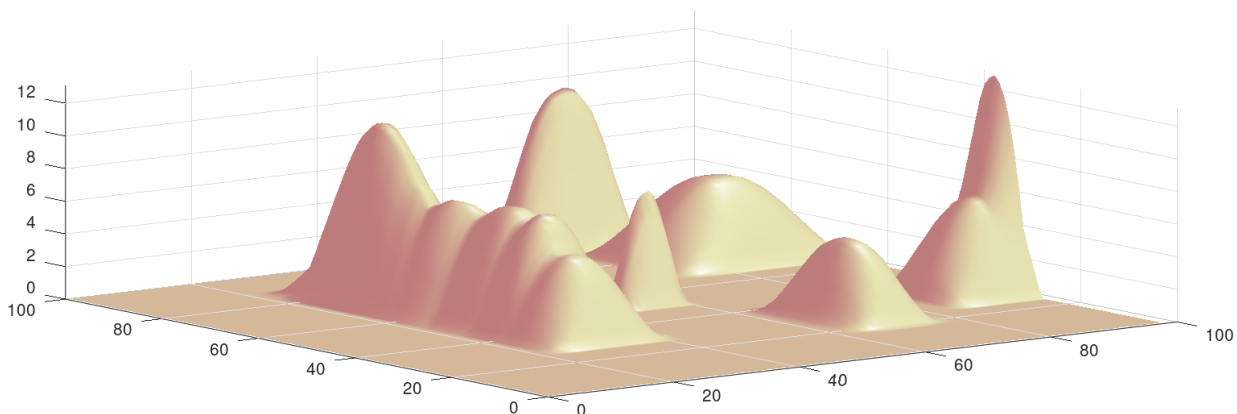
## Underwater Environment

### 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(
2     ScattererClass<double>& scatterers
3 ){
4
5     // auto save_files {false};
6     const auto save_files {false};
7     const auto hill_creation_flag {true};
8
9     // sea-floor bounds
10    auto bed_width {100.00};
11    auto bed_length {100.00};
12
13    // creating tensors for coordinates and reflectivity
14    vector<vector<double>> box_coordinates;
15    vector<double> box_reflectivity;
16
17    // scatter density
18    // auto bed_width_density {static_cast<double>( 10.00)};
19    // auto bed_length_density {static_cast<double>( 10.00)};
20    auto bed_width_density {static_cast<double>( 5.00)};
21    auto bed_length_density {static_cast<double>( 5.00)};
22
23    // setting up coordinates
24    auto xpoints {svr::linspace<double>(
25        0.00,
26        bed_width,
27        bed_width * bed_width_density
28    )};
29    auto ypoints {svr::linspace<double>(
30        0.00,
31        bed_length,
32        bed_length * bed_length_density
33    )};
34    if(save_files) fWriteVector(xpoints, "../csv-files/xpoints.csv"); // verified
35    if(save_files) fWriteVector(ypoints, "../csv-files/ypoints.csv"); // verified
36
37    // creating mesh
38    auto [xgrid, ygrid] = meshgrid(std::move(xpoints), std::move(ypoints));
39    if(save_files) fWriteMatrix(xgrid, "../csv-files/xgrid.csv"); // verified
40    if(save_files) fWriteMatrix(ygrid, "../csv-files/ygrid.csv"); // verified
41
42    // reshaping
43    auto X {reshape(xgrid, xgrid.size()*xgrid[0].size())};
44    auto Y {reshape(ygrid, ygrid.size()*ygrid[0].size())};
45    if(save_files) fWriteVector(X, "../csv-files/X.csv"); // verified
46    if(save_files) fWriteVector(Y, "../csv-files/Y.csv"); // verified
47
48    // creating heights of scatterers
49    if(hill_creation_flag){

```

```

50
51 // setting up hill parameters
52 auto    num_hills    {10};
53
54 // setting up placement of hills
55 auto    points2D      {concatenate<0>(X, Y)};           // verified
56 auto    min2D          {min<1, double>(points2D)};      // verified
57 auto    max2D          {max<1, double>(points2D)};      // verified
58 auto    hill_2D_center {min2D + \
59                      rand({2, num_hills}) * (max2D - min2D)}; // verified
60
61 // setup: hill-dimensions
62 auto    hill_dimensions_min {transpose(vector<double>{5, 5, 2})}; // verified
63 auto    hill_dimensions_max {transpose(vector<double>{30, 30, 10})}; // verified
64 auto    hill_dimensions     {hill_dimensions_min + \
65                      rand({3, num_hills}) * (hill_dimensions_max -
66                      hill_dimensions_min)};           // verified
67
68 // function-call: hill-creation function
69 fCreateHills(hill_2D_center,
70             hill_dimensions,
71             points2D);
72
73 // setting up floor reflectivity
74 auto    floorScatter_reflectivity {std::vector<double>(Y.size(), 1.00)};
75
76 // populating the values of the incoming argument
77 scatterers.coordinates = std::move(points2D);
78 scatterers.reflectivity = std::move(floorScatter_reflectivity);
79
80 else{
81
82 // assigning flat heights
83 auto    Z {std::vector<double>(Y.size(), 0)};
84
85 // setting up floor coordinates
86 auto    floorScatter_coordinates {concatenate<0>(X, Y, Z)};
87 auto    floorScatter_reflectivity {std::vector<double>(Y.size(), 1)};
88
89 // populating the values of the incoming argument
90 scatterers.coordinates = std::move(floorScatter_coordinates);
91 scatterers.reflectivity = std::move(floorScatter_reflectivity);
92
93 }
94
95 // printing status
96 std::cout << format("> Finished Sea-Floor Setup \n");
97 }

```

---

# Chapter 2

## Transmitter

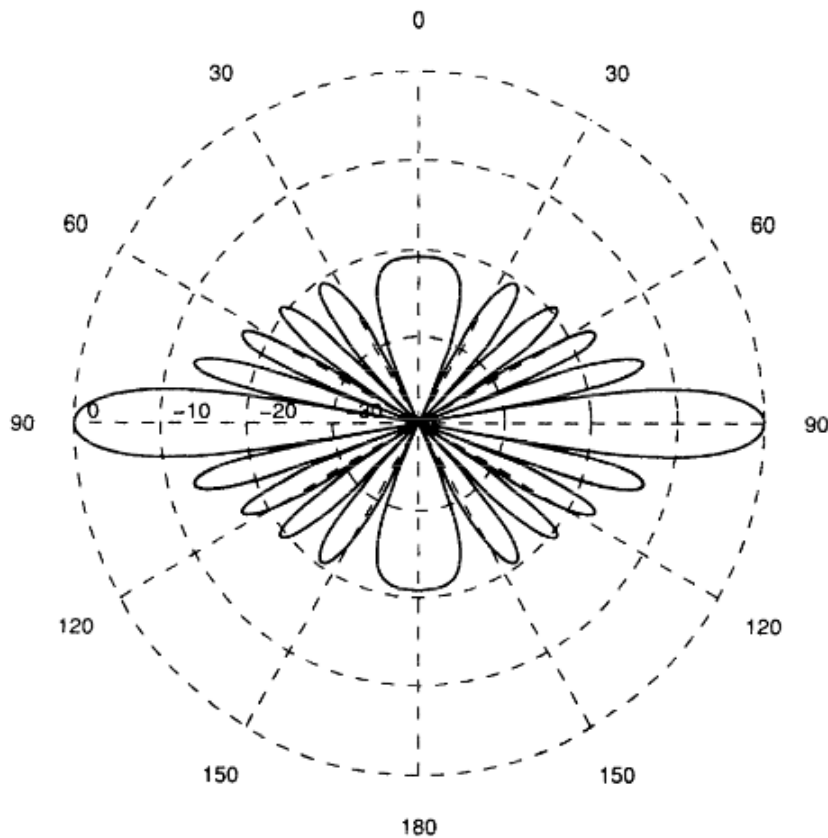


Figure 2.1: Beampattern of a Transmission Uniform Linear Array

### 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

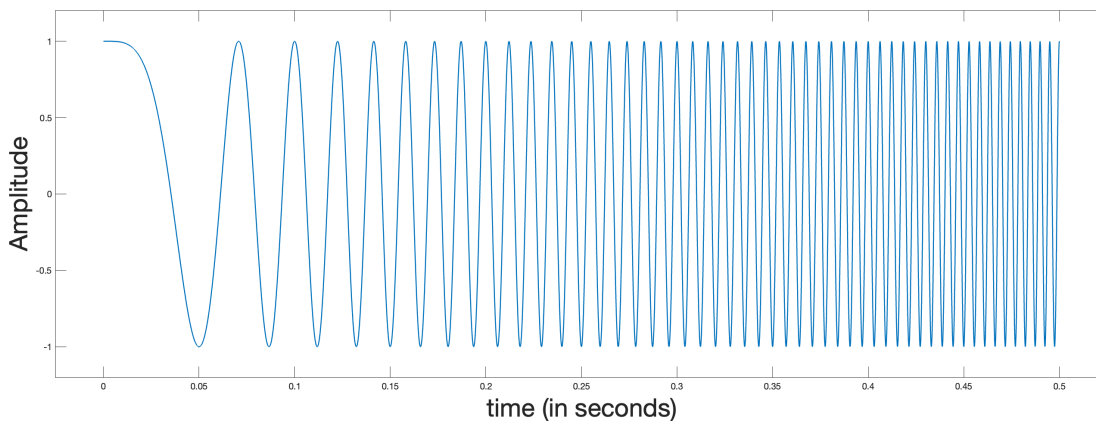


Figure 2.2: Linear Frequency Modulated Wave

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     int               elevationQuantDensity; // quantization of angles along the
30     T                 rangeQuantSize;      // range-cell size when shadowing
31     T                 azimuthShadowThreshold; // azimuth thresholding
32     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 <
2      typename T,
3      typename = std::enable_if_t<
4          std::is_same_v<T, double> ||
5          std::is_same_v<T, float>
6      >
7  >
8  void fTransmitterSetup(
9      TransmitterClass<T>& transmitter_fls,
10     TransmitterClass<T>& transmitter_portside,
11     TransmitterClass<T>& transmitter_starboard
12 ){
13     // Setting up transmitter
14     T sampling_frequency {160e3}; // sampling frequency
15     T f1 {50e3}; // first frequency of LFM
16     T f2 {70e3}; // second frequency of LFM
17     T fc {(f1 + f2)/2.00}; // finding center-frequency
18     T bandwidth {std::abs(f2 - f1)}; // bandwidth
19     T pulselength {5e-2}; // time of recording
20
21     // building LFM
22     auto timearray {svr::linspace<T>(
23         0.00,
24         pulselength,
25         std::floor(pulselength * sampling_frequency)
26     )};
27     auto K {f2 - f1/pulselength}; // calculating frequency-slope
28     auto Signal {cos(2 * std::numbers::pi * \
29         (f1 + K*timearray) * \

```

```

30         timearray));           // frequency at each time-step, with f1
31         = 0
32 // Setting up transmitter
33 auto location = std::vector<T>(3, 0); // location of
34     transmitter
35 T azimuthal_angle_fls = {0}; // initial
36     pointing direction
37 T azimuthal_angle_port = {90}; // initial
38     pointing direction
39 T azimuthal_angle_starboard = {-90}; // initial
40     pointing direction
41 T elevation_angle = {-60}; // initial
42     pointing direction
43
44 T azimuthal_beamwidth_fls = {20}; // azimuthal
45     beamwidth of the signal cone
46 T azimuthal_beamwidth_port = {20}; // azimuthal
47     beamwidth of the signal cone
48 T azimuthal_beamwidth_starboard = {20}; // azimuthal
49     beamwidth of the signal cone
50
51 T elevation_beamwidth_fls = {20}; // elevation
52     beamwidth of the signal cone
53 T elevation_beamwidth_port = {20}; // elevation
54     beamwidth of the signal cone
55 T elevation_beamwidth_starboard = {20}; // elevation
56     beamwidth of the signal cone
57
58 int azimuthQuantDensity = {10}; // number of points, a degree is split
59     into quantization density along azimuth (used for shadowing)
60 int elevationQuantDensity = {10}; // number of points, a degree is split
61     into quantization density along elevation (used for shadowing)
62 T rangeQuantSize = {10}; // the length of a cell (used for
63     shadowing)
64
65 T azimuthShadowThreshold = {1}; // azimuth threshold (in degrees)
66 T elevationShadowThreshold = {1}; // elevation threshold (in degrees)
67
68 // transmitter-fls
69 transmitter_fls.location = location; // Assigning
70     location
71 transmitter_fls.signal = Signal; // Assigning
72     signal
73 transmitter_fls.azimuthal_angle = azimuthal_angle_fls; // assigning
74     azimuth angle
75 transmitter_fls.elevation_angle = elevation_angle; // assigning
76     elevation angle
77 transmitter_fls.azimuthal_beamwidth = azimuthal_beamwidth_fls; // assigning
78     azimuth-beamwidth
79 transmitter_fls.elevation_beamwidth = elevation_beamwidth_fls; // assigning
80     elevation-beamwidth
81 // updating quantization densities
82 transmitter_fls.azimuthQuantDensity = azimuthQuantDensity; // assigning
83     azimuth quant density
84 transmitter_fls.elevationQuantDensity = elevationQuantDensity; // assigning
85     elevation quant density

```

```

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

```



```
104 transmitter_starboard.azimuthQuantDensity    = azimuthQuantDensity;    //
      assigning azimuth-quant-density
105 transmitter_starboard.elevationQuantDensity  = elevationQuantDensity;
106 transmitter_starboard.rangeQuantSize         = rangeQuantSize;
107 transmitter_starboard.azimuthShadowThreshold = azimuthShadowThreshold;
108 transmitter_starboard.elevationShadowThreshold = elevationShadowThreshold;
109 // signal related
110 transmitter_starboard.f_low                   = f1;                      //
      assigning lower frequency
111 transmitter_starboard.f_high                  = f2;                      //
      assigning higher frequency
112 transmitter_starboard.fc                     = fc;                      //
      assigning center frequency
113 transmitter_starboard.bandwidth               = bandwidth;              //
      assigning bandwidth
114
115 }
```

---

# Chapter 3

## Uniform Linear Array

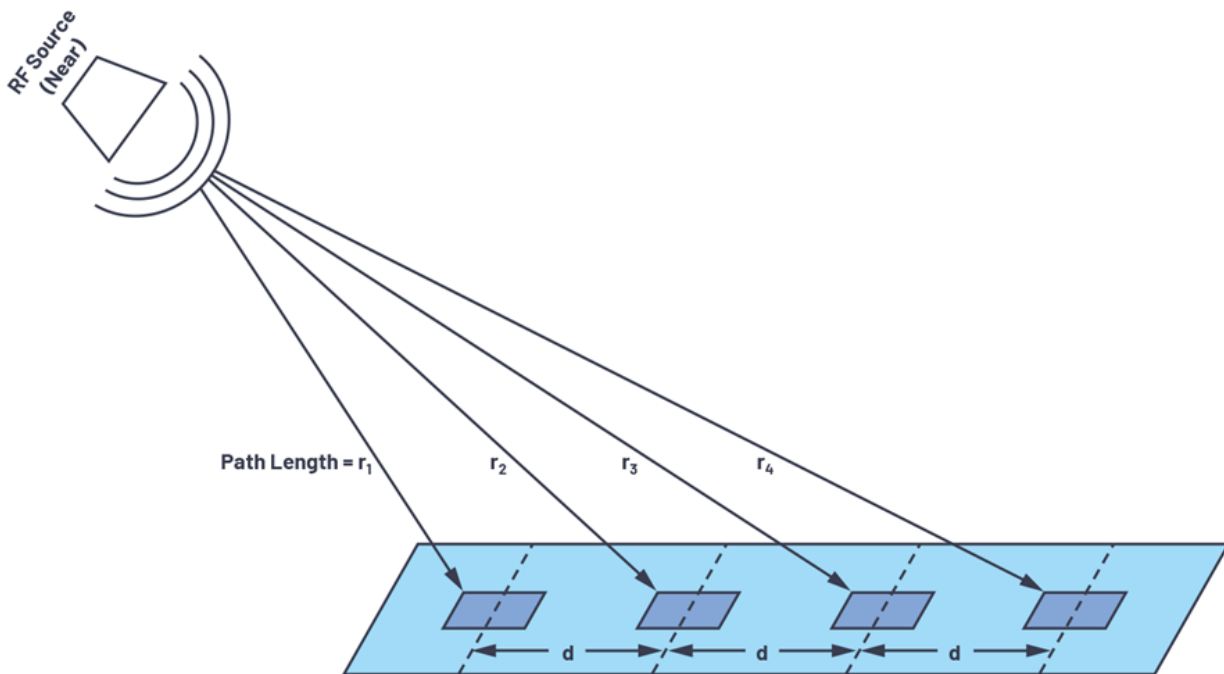


Figure 3.1: Uniform Linear Array

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

```

```

31  std::vector<std::vector<complex<T>>> mulFFTMMatrix; // the matrix containing the
    delays for each-element as a slot
32  std::vector<complex<T>> matchFilter; // torch tensor containing the
    match-filter
33  int num_buffer_zeros_per_frame; // number of zeros we're adding
    per frame to ensure no-rotation
34  std::vector<std::vector<T>> beamformedImage; // the beamformed image
35  std::vector<std::vector<T>> cartesianImage; // the cartesian version of
    beamformed image
36
37  // Artificial acoustic-image related
38  std::vector<std::vector<T>> currentArtificialAcousticImage; // acoustic image
    directly produced
39
40
41  // Basic Constructor
42  ULAClass() = default;
43
44  // constructor
45  ULAClass(const int num_sensors_arg,
46            const auto inter_element_spacing_arg,
47            const auto& coordinates_arg,
48            const auto& sampling_frequency_arg,
49            const auto& recording_period_arg,
50            const auto& location_arg,
51            const auto& signalMatrix_arg,
52            const auto& lowpass_filter_coefficients_for_decimation_arg):
53      num_sensors(num_sensors_arg),
54      inter_element_spacing(inter_element_spacing_arg),
55      coordinates(std::move(coordinates_arg)),
56      sampling_frequency(sampling_frequency_arg),
57      recording_period(recording_period_arg),
58      location(std::move(location_arg)),
59      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 <
2     typename T,
3     typename = std::enable_if_t<
4         std::is_same_v<T, double> ||
5         std::is_same_v<T, float>
6     >
7 >
8 void fULASetup(
9     ULAClass<T>&    ula_fls,
10    ULAClass<T>&    ula_portside,
11    ULAClass<T>&    ula_starboard)
12 {
13     // setting up ula
14     auto num_sensors          {static_cast<int>(16)};           // number of sensors
15     T    sampling_frequency   {static_cast<T>(160e3)};         // sampling frequency
16     T    inter_element_spacing {1500/(2*sampling_frequency)}; // space between
17     samples
18     T    recording_period     {10e-2};                         // sampling-period
19
20     // building the direction for the sensors
21     auto ULA_direction        {std::vector<T>({-1, 0, 0})};
22     auto ULA_direction_norm    {norm(ULA_direction)};
23     if (ULA_direction_norm != 0) {ULA_direction = ULA_direction/ULA_direction_norm;}
24     ULA_direction              = ULA_direction * inter_element_spacing;
25
26     // building coordinates for sensors
27     auto ULA_coordinates      {
28         transpose(ULA_direction) * \
29         svr::linspace<double>(
30             0.00,
31             num_sensors -1,
32             num_sensors)
33     };
34
35     // coefficients of decimation filter
36     auto lowpassfiltercoefficients {std::vector<T>{0.0000, 0.0000, 0.0000, 0.0000,
37         0.0000, 0.0000, 0.0001, 0.0003, 0.0006, 0.0015, 0.0030, 0.0057, 0.0100, 0.0163,
38         0.0251, 0.0364, 0.0501, 0.0654, 0.0814, 0.0966, 0.1093, 0.1180, 0.1212, 0.1179,
39         0.1078, 0.0914, 0.0699, 0.0451, 0.0192, -0.0053, -0.0262, -0.0416, -0.0504,
40         -0.0522, -0.0475, -0.0375, -0.0239, -0.0088, 0.0057, 0.0179, 0.0263, 0.0303,
41         0.0298, 0.0253, 0.0177, 0.0086, -0.0008, -0.0091, -0.0153, -0.0187, -0.0191,
42         -0.0168, -0.0123, -0.0065, -0.0004, 0.0052, 0.0095, 0.0119, 0.0125, 0.0112,
43         0.0084, 0.0046, 0.0006, -0.0031, -0.0060, -0.0078, -0.0082, -0.0075, -0.0057,

```

```

-0.0033, -0.0006, 0.0019, 0.0039, 0.0051, 0.0055, 0.0050, 0.0039, 0.0023, 0.0005,
-0.0012, -0.0025, -0.0034, -0.0036, -0.0034, -0.0026, -0.0016, -0.0004, 0.0007,
0.0016, 0.0022, 0.0024, 0.0023, 0.0018, 0.0011, 0.0003, -0.0004, -0.0011,
-0.0015, -0.0016, -0.0015}};

```

```

36
37 // assigning values
38 ula_fls.num_sensors = num_sensors; //
    assigning number of sensors
39 ula_fls.inter_element_spacing = inter_element_spacing; //
    assigning inter-element spacing
40 ula_fls.coordinates = ULA_coordinates; //
    assigning ULA coordinates
41 ula_fls.sampling_frequency = sampling_frequency; //
    assigning sampling frequencys
42 ula_fls.recording_period = recording_period; //
    assigning recording period
43 ula_fls.sensor_direction = ULA_direction; // ULA
    direction
44 ula_fls.lowpass_filter_coefficients_for_decimation = lowpassfiltercoefficients; //
    storing coefficients
45
46
47 // assigning values
48 ula_portside.num_sensors = num_sensors; //
    assigning number of sensors
49 ula_portside.inter_element_spacing = inter_element_spacing; //
    assigning inter-element spacing
50 ula_portside.coordinates = ULA_coordinates; //
    assigning ULA coordinates
51 ula_portside.sampling_frequency = sampling_frequency; //
    assigning sampling frequencys
52 ula_portside.recording_period = recording_period; //
    assigning recording period
53 ula_portside.sensor_direction = ULA_direction; //
    ULA direction
54 ula_portside.lowpass_filter_coefficients_for_decimation = lowpassfiltercoefficients;
    // storing coefficients
55
56
57 // assigning values
58 ula_starboard.num_sensors = num_sensors; //
    assigning number of sensors
59 ula_starboard.inter_element_spacing = inter_element_spacing; //
    assigning inter-element spacing
60 ula_starboard.coordinates = ULA_coordinates; //
    assigning ULA coordinates
61 ula_starboard.sampling_frequency = sampling_frequency; //
    assigning sampling frequencys
62 ula_starboard.recording_period = recording_period; //
    assigning recording period
63 ula_starboard.sensor_direction = ULA_direction; //
    ULA direction
64 ula_starboard.lowpass_filter_coefficients_for_decimation =
    lowpassfiltercoefficients; // storing coefficients
65 }

```

---

# Chapter 4

## Autonomous Underwater Vehicle

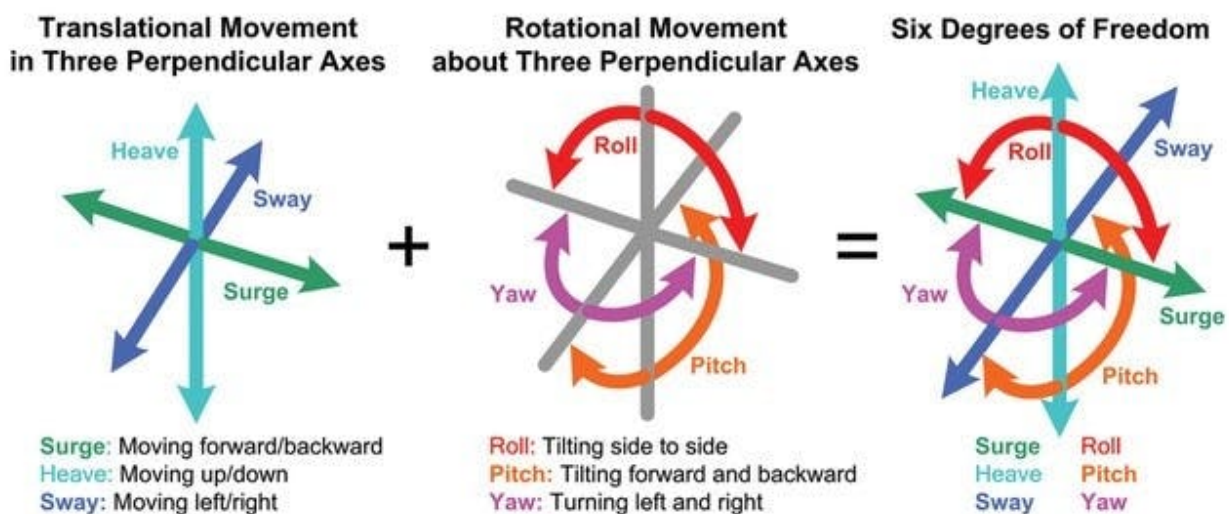


Figure 4.1: AUV degrees of freedom

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

```



```

31 // spotlight mode related
32 std::vector<std::vector<T>> absolute_coords_patch_cart; // cartesian coordinates of
    patch
33
34 // Synthetic Aperture Related
35 std::vector<std::vector<T>> ApertureSensorLocations; // sensor locations of
    aperture
36
37 // functions
38 void syncComponentAttributes();
39 void init(svr::ThreadPool& thread_pool);
40 void simulate_signal(
41     const ScattererClass<T>&          seafloor,
42     svr::ThreadPool&                  thread_pool,
43     svr::FFTPlanUniformPoolHandle<T, std::complex<T>>& fft_pool_handle,
44     svr::IFFTPlanUniformPoolHandle<std::complex<T>, T>& ifft_pool_handle
45 );
46 void subset_scatterers(
47     const ScattererClass<T>&          seafloor,
48     svr::ThreadPool&                  thread_pool,
49     std::vector<std::size_t>&         fls_scatterer_indices,
50     std::vector<std::size_t>&         portside_scatterer_indices,
51     std::vector<std::size_t>&         starboard_scatterer_indices
52 );

```

---

## 4.2 AUV Setup Scripts

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

---

```

1  template <
2      typename T,
3      typename = std::enable_if_t<
4          std::is_same_v<T, double> ||
5          std::is_same_v<T, float>
6      >
7  >
8  void fAUVSetup(AUVClass<T>& auv) {
9
10     // building properties for the auv
11     auto location {std::vector<T>{0, 50, 30}}; // starting location
12     auto velocity {std::vector<T>{5, 0, 0}}; // starting velocity
13     auto pointing_direction {std::vector<T>{1, 0, 0}}; // pointing direction
14
15     // assigning
16     auv.location = std::move(location); // assigning location
17     auv.velocity = std::move(velocity); // assigning velocity
18     auv.pointing_direction = std::move(pointing_direction); // assigning pointing
        direction
19
20     // signaling end
21     std::cout << format("> Completed AUV-setup\n");
22
23 }

```

---

## **Part II**

# **Signal Simulation Pipeline**

# Chapter 5

## Signal Simulation

### Overview

The signal simulation pipeline is the pipeline responsible for simulating/modeling the signals sampled by the ULA-sensors under a real sub-marine environment. This chapter, and the subsequent ones, deal with the assumptions, mathematics, physics and code that goes into the design and creation of the pipeline.

A disclaimer that goes without saying is that signal-simulation is a world of its own. There's a reason that comsol, flexcompute and other numerical-simulation based companies exist. To write a signal simulation, from scratch, while these entities exist, and to make any case that this competes with those, would be flirting with delusion.

To that end, we don't write general-purpose signal simulation pipeline. However, the effort in the signal-simulator direction is purely for application-specific reasons. This is something I can talk about. One of the major in-house signal simulation pipelines yours truly developed at Naval Physical and Oceanographic Laboratory did just that. The aim of that pipeline was not to re-invent the wheel. But to create one that existed at the right speed-fidelity trade-off that the institute operated in. The pipeline created during my time there had several toggles corresponding to the different information to consider during simulation. The more information pertaining to the environment, is involved, the higher the compute and time required. Thus, mid-to-high fidelity pipelines often involve writing well-tuned GPU-supported C++ (think, CUDA). And this is important when you have pipelines downstream whose outputs depend on the signal accuracy, and by association, signal simulator fidelity.

To that end, understanding what this pipeline is not, is perhaps just as important as what it is. The core priority of this signal simulator pipeline is to produce signals for navigation. Navigation does not require high-accuracy signals owing to the very simple fact that decisions made from high-accuracy signals and low-accuracy signals tend to be the same as long as environment-topology information is not lost. To grossly oversimplify what I mean by that, the outcome of your driving doesn't change whether you have high-definition LIDAR mapping the surrounding environment to the millimeter level or if you're just driving with your eyes. Thus fidelity of simulator is not a priority and I will not be putting in the kind of effort I put in at NPOL, for this reason (also because I don't want OPSEC to be

mad).

To put it simply, the signal simulation pipeline is quite trivial as far as signal simulators are concerned. But it'll work perfectly for our purposes. And thus, we'll be choosing the simplest of systems and one I like to call, "the EE engineer's best friend": the infamous Linear Time Invariant systems.

## **Part III**

# **Imaging Pipeline**

## **Part IV**

# **Perception & Control Pipeline**

# Appendix A

## Application Specific Tools

### A.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  }

```

---

## A.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 }

```

---

## A.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          if(in_   != nullptr) {fftw_free(           in_);}
29          if(out_  != nullptr) {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
41          // allocating nfft
42          this->nfft_ = nfft;
43
44          // allocating input-region
45          in_ = reinterpret_cast<fftw_complex*>(
46              fftw_malloc(nfft_ * sizeof(fftw_complex))
47          );
48          out_ = reinterpret_cast<fftw_complex*>(
49              fftw_malloc(nfft_ * sizeof(fftw_complex))
50          );
51          if(!in_ || !out_) {throw std::runtime_error("FILE: FFTPlanClass.hpp |
52                      CLASS: FFTPlanClass | REPORT: in-out allocation failed");}
53
54          // creating plan
55          plan_ = fftw_plan_dft_1d(
56              static_cast<int>(nfft_),
57              in_,
58              out_,
59              FFTW_FORWARD,

```

```

58         FFTW_MEASURE
59     );
60     if(!plan_) {throw std::runtime_error("FILE: FFTPlanClass.hpp |
61         CLASS: FFTPlanClass | REPORT: plan-creation failed");}
62 }
63 /*=====
64 Copy Constructor
65 -----*/
66 FFTPlanClass(const FFTPlanClass& other)
67 {
68     // copying nfft
69     nfft_ = other.nfft_;
70     cout << format("\t\t FFTPlanClass(const FFTPlanClass& other) | nfft_ =
71         {}\n", nfft_);
72
73     // allocating input-region
74     in_ = reinterpret_cast<fftw_complex*>(
75         fftw_malloc(nfft_ * sizeof(fftw_complex))
76     );
77     out_ = reinterpret_cast<fftw_complex*>(
78         fftw_malloc(nfft_ * sizeof(fftw_complex))
79     );
80     if(!in_ || !out_) {throw std::runtime_error("FILE: FFTPlanClass.hpp |
81         CLASS: FFTPlanClass | REPORT: in-out allocation failed");}
82
83     // copying input-region and output-region
84     std::memcpy(in_, other.in_, nfft_ * sizeof(fftw_complex));
85     std::memcpy(out_, other.out_, nfft_ * sizeof(fftw_complex));
86
87     // creating plan
88     plan_ = fftw_plan_dft_1d(
89         static_cast<int>(nfft_),
90         in_,
91         out_,
92         FFTW_FORWARD,
93         FFTW_MEASURE
94     );
95     if(!plan_) {throw std::runtime_error("FILE: FFTPlanClass.hpp |
96         CLASS: FFTPlanClass | REPORT: plan-creation failed");}
97 }
98 /*=====
99 Copy Assignment
100 -----*/
101 FFTPlanClass& operator=(const FFTPlanClass& other)
102 {
103     // handling self-assignment
104     if (this == &other) {return *this;}
105
106     // cleaning-up existing resources
107     if(plan_ != nullptr) {fftw_destroy_plan( plan_);}
108     if(in_ != nullptr) {fftw_free( in_);}
109     if(out_ != nullptr) {fftw_free( out_);}
110
111     // allocating input-region and output-region
112     nfft_ = other.nfft_;
113     in_ = reinterpret_cast<fftw_complex*>(
114         fftw_malloc(nfft_ * sizeof(fftw_complex))
115     );
116     out_ = reinterpret_cast<fftw_complex*>(

```

```

113         fftw_malloc(nfft_ * sizeof(fftw_complex))
114     );
115     if(!in_ || !out_) {throw std::runtime_error("FILE: FFTPlanClass.hpp |
116         CLASS: FFTPlanClass | FUNCTION: Copy-Assignment | REPORT: in-out
117         allocation failed");}
118
119     // copying contents
120     cout << format("\t\t FFTPlanClass& operator=(const FFTPlanClass& other) |
121         nfft_ = {}\n", nfft_);
122     std::memcpy(in_, other.in_, nfft_ * sizeof(fftw_complex));
123     std::memcpy(out_, other.out_, nfft_ * sizeof(fftw_complex));
124
125     // creating engine
126     plan_ = fftw_plan_dft_id(
127         static_cast<int>(nfft_),
128         in_,
129         out_,
130         FFTW_FORWARD,
131         FFTW_MEASURE
132     );
133     if(!plan_) {throw std::runtime_error("FILE: FFTPlanClass.hpp | CLASS:
134         FFTPlanClass | FUNCTION: Copy-Assignment | REPORT: plan-creation
135         failed");}
136
137     // returning
138     return *this;
139 }
140
141 /*=====
142 Move Constructor
143 -----*/
144 FFTPlanClass(FFTPlanClass&& other)
145 :   nfft_(      other.nfft_),
146     in_(        other.in_),
147     out_(        other.out_),
148     plan_(       other.plan_)
149 {
150     // resetting the others
151     other.nfft_   = 0;
152     other.in_     = nullptr;
153     other.out_    = nullptr;
154     other.plan_   = nullptr;
155 }
156
157 /*=====
158 Move Assignment
159 -----*/
160 FFTPlanClass& operator=(FFTPlanClass&& other)
161 {
162     // self-assignment check
163     if (this == &other) {return *this;}
164
165     // cleaning up existing resources
166     if(plan_ != nullptr) {fftw_destroy_plan(   plan_);}
167     if(in_   != nullptr) {fftw_free(           in_);}
168     if(out_  != nullptr) {fftw_free(           out_);}
169
170     // Copying-values and changing pointers
171     nfft_ = other.nfft_;
172     cout << format("\t\t FFTPlanClass's MOVE assignment | nfft_ = {}\n",
173         nfft_);

```

```

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

```

```

223         );
224     }
225     else if constexpr(
226         std::is_same_v< destinationType, double >
227     ){
228         output_vector[index] = std::sqrt(
229             std::pow(out_[index][0], 2) + \
230             std::pow(out_[index][1], 2)
231         );
232     }
233 }
234
235 // returning output
236 return std::move(output_vector);
237 }
238 /*=====
239 Running fft - balanced
240 -----*/
241 std::vector<destinationType>
242 fft_l2_conserved(const std::vector<sourceType>& input_vector)
243 {
244     // throwing an error
245     if (input_vector.size() > nfft_)
246         throw std::runtime_error("FILE: FFTPlanClass.hpp | CLASS: FFTPlanClass
247             | FUNCTION: fft() | REPORT: input-vector size is greater than
248             NFFT");
249
250     // copying inputs
251     for(std::size_t index = 0; index < input_vector.size(); ++index)
252     {
253         if constexpr(
254             std::is_same_v< sourceType, double >
255         ){
256             in_[index][0] = input_vector[index];
257             in_[index][1] = 0;
258         }
259         else if constexpr(
260             std::is_same_v< sourceType, std::complex<double> >
261         ){
262             in_[index][0] = input_vector[index].real();
263             in_[index][1] = input_vector[index].imag();
264         }
265     }
266
267     // executing fft
268     fftw_execute(plan_);
269
270     // copying results to output-vector
271     std::vector<destinationType> output_vector(nfft_);
272     for(std::size_t index = 0; index < nfft_; ++index)
273     {
274         if constexpr(
275             std::is_same_v< destinationType, std::complex<double> >
276         ){
277             output_vector[index] = std::complex<double>(
278                 out_[index][0] * (1.00 / std::sqrt(nfft_)),
279                 out_[index][1] * (1.00 / std::sqrt(nfft_))
280             );
281         }
282     }
283 }

```

```

280         else if constexpr(
281             std::is_same_v< destinationType, double >
282         ){
283             output_vector[index] = std::sqrt(
284                 std::pow(out_[index][0] * (1.00 / std::sqrt(nfft_)), 2) + \
285                 std::pow(out_[index][1] * (1.00 / std::sqrt(nfft_)), 2)
286             );
287         }
288     }
289
290     // returning output
291     return std::move(output_vector);
292 }
293 };
294 }

```

---

## A.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             if(plan_ != nullptr) {fftw_destroy_plan( plan_);}
23             if(in_ != nullptr) {fftw_free( in_);}
24             if(out_ != nullptr) {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     };
40 }

```

```

35
36 // 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     if(plan_ != nullptr) {fftw_destroy_plan(plan_);}
88     if(in_ != nullptr) {fftw_free(in_);}
89     if(out_ != nullptr) {fftw_free(out_);}
90
91     // allocating space
92     nfft_ = other.nfft_;

```



```

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

```

```

140         out_      = other.out_;
141         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     }

```

```

198     }
199
200     // returning
201     return std::move(output_vector);
202 }
203
204 //=====
205 Running - proper bases change
206 -----*/
207
208 std::vector<destinationType>
209 ifft_l2_conserved(const std::vector<sourceType>& input_vector)
210 {
211     // throwing an error
212     if (input_vector.size() > nfft_)
213         throw std::runtime_error("File: FFTPlanClass | Class: IFFTPlanClass |
214             Function: ifft() | Report: size of vector > nfft ");
215
216     // copy input into fftw buffer
217     for(std::size_t index = 0; index < nfft_; ++index)
218     {
219         if constexpr(
220             std::is_same_v< sourceType, std::complex<double> >
221         ){
222             in_[index][0] = input_vector[index].real();
223             in_[index][1] = input_vector[index].imag();
224         }
225         else if constexpr(
226             std::is_same_v< sourceType, double >
227         ){
228             in_[index][0] = input_vector[index];
229             in_[index][1] = 0;
230         }
231     }
232
233     // execute ifft
234     fftw_execute(plan_);
235
236     // normalize output
237     std::vector<destinationType> output_vector(nfft_);
238     for(std::size_t index = 0; index < nfft_; ++index)
239     {
240         if constexpr(
241             std::is_same_v< destinationType, double >
242         ){
243             output_vector[index] = out_[index][0] * 1.00/std::sqrt(nfft_);
244         }
245         else if constexpr(
246             std::is_same_v< destinationType, std::complex<double> >
247         ){
248             output_vector[index][0] = std::complex<double>(
249                 out_[index][0] * 1.00/std::sqrt(nfft_),
250                 out_[index][1] * 1.00/std::sqrt(nfft_)
251             );
252         }
253     }
254
255     // returning
256     return std::move(output_vector);
257 }
258
259 };

```

256 }

## A.5 FFT Plan Pool

```

1  #pragma once
2  namespace svr {
3
4      template <
5          typename sourceType,
6          typename destinationType,
7          typename = std::enable_if_t<
8              std::is_same_v<sourceType, double> &&
9              std::is_same_v<destinationType, std::complex<double>>
10         >
11     >
12     class FFTPlanUniformPool {
13     public:
14         /*=====
15         Handle to Plan
16         -----*/
17         struct AccessPairs
18         {
19             /*=====
20             Members
21             -----*/
22             svr::FFTPlanClass<sourceType, destinationType>& plan;
23             std::unique_lock<std::mutex> lock;
24
25             /*=====
26             Special Members
27             -----*/
28             AccessPairs() = delete;
29             AccessPairs(
30                 svr::FFTPlanClass<sourceType, destinationType>& plan_arg,
31                 std::mutex& plan_mutex
32             ) : plan(plan_arg), lock(plan_mutex) {}
33             AccessPairs(
34                 svr::FFTPlanClass<sourceType, destinationType>& plan_arg,
35                 std::unique_lock<std::mutex>&& lock_arg
36             ) : plan(plan_arg), lock(std::move(lock_arg)) {}
37             AccessPairs(const AccessPairs& other) = delete;
38             AccessPairs& operator=(const AccessPairs& other) = delete;
39             AccessPairs(AccessPairs&& other) = delete;
40             AccessPairs& operator=(AccessPairs&& other) = delete;
41         };
42
43         /*=====
44         Core Members
45         -----*/
46         std::vector<svr::FFTPlanClass<sourceType, destinationType>> plans;
47         std::vector<std::mutex> mutexes;
48
49         /*=====
50         Special-Members
51         -----*/
52         FFTPlanUniformPool() = default;

```

```

53     FFTPlanUniformPool(const std::size_t    num_plans,
54                        const std::size_t    nfft)
55     {
56         // reserving space
57         plans.reserve(num_plans);
58         for(auto i = 0; i < num_plans; ++i){
59             plans.emplace_back(nfft);
60         }
61
62         // creating a vector of mutexes
63         mutexes = std::move(std::vector<std::mutex>(num_plans));
64     }
65     FFTPlanUniformPool(const FFTPlanUniformPool& other)      = delete;
66     FFTPlanUniformPool& operator=(const FFTPlanUniformPool& other) = delete;
67     FFTPlanUniformPool(FFTPlanUniformPool&& other)           = default;
68     FFTPlanUniformPool& operator=(FFTPlanUniformPool&& other) = default;
69
70     /*=====
71     Function to fetch a plan
72         > searches for a free-plan
73         > if found, locks the plan
74         > return the handle to the plan
75     -----*/
76     AccessPairs fetch_plan() {
77         const int num_rounds = 12;
78         for (int round = 0; round < num_rounds; ++round) {
79             for (int i = 0; i < mutexes.size(); ++i) {
80
81                 std::unique_lock<std::mutex> curr_lock(
82                     mutexes[i],
83                     std::try_to_lock
84                 );
85                 if (curr_lock.owns_lock())
86                     return AccessPairs(plans[i], std::move(curr_lock));
87             }
88         }
89         throw std::runtime_error(
90             "FILE: FFTPlanPoolClass.hpp | CLASS: FFTPlanUniformPool | FUNCTION:
91             fetch_plan() | "
92             "Report: No plans available despite num_rounds rounds of searching");
93     }
94 };
95 }

```

---

## A.6 IFFT Plan Pool

```

1  #pragma once
2
3  /*=====
4  Dependencies
5  -----*/
6
7  namespace svr {
8
9      template <

```

```

10     typename    sourceType,
11     typename    destinationType,
12     typename    =    std::enable_if_t<
13         std::is_same_v<sourceType, std::complex<double>>&&
14         std::is_same_v<destinationType, double>
15     >
16 >
17 class IFFTPlanUniformPool
18 {
19     public:
20         /*=====
21         Structure used for interfacing to plans
22         -----*/
23         struct AccessPairs
24         {
25             /*=====
26             Core Members
27             -----*/
28             svr::IFFTPlanClass<sourceType, destinationType>&    plan;
29             std::unique_lock<std::mutex>                        lock;
30
31             /*=====
32             Special Members
33             -----*/
34             AccessPairs()                                     =    delete;
35             AccessPairs(
36                 svr::IFFTPlanClass<sourceType, destinationType>& plan_arg,
37                 std::mutex&                                plan_mutex_arg
38             ): plan(plan_arg), lock(plan_mutex_arg) {}
39             AccessPairs(
40                 svr::IFFTPlanClass<sourceType, destinationType>& plan_arg,
41                 std::unique_lock<std::mutex>&&                lock_arg
42             ): plan(plan_arg), lock(std::move(lock_arg)) {}
43             AccessPairs(const AccessPairs&    other)          =    delete;
44             AccessPairs&    operator=(const AccessPairs& other) =    delete;
45             AccessPairs(AccessPairs&& other)                  =    delete;
46             AccessPairs&    operator=(AccessPairs&& other)    =    delete;
47         };
48
49         /*=====
50         Core Members
51         -----*/
52         std::vector<    svr::IFFTPlanClass<sourceType, destinationType> > plans;
53         std::vector<    std::mutex                                     >    mutexes;
54
55         /*=====
56         Special Members
57         -----*/
58         IFFTPlanUniformPool()                                     =    default;
59         IFFTPlanUniformPool(const std::size_t num_plans,
60                             const    std::size_t nfft)
61         {
62             // reserving space
63             plans.reserve(num_plans);
64             for(auto i = 0; i < num_plans; ++i)
65                 plans.emplace_back(nfft);
66
67             // creating vector of mutexes
68             mutexes    =    std::vector<std::mutex>(num_plans);

```

```

69     }
70     IFFTPlanUniformPool(const IFFTPlanUniformPool& other)           = delete;
71     IFFTPlanUniformPool& operator=(const IFFTPlanUniformPool& other) = delete;
72     IFFTPlanUniformPool(IFFTPlanUniformPool&& other)               = default;
73     IFFTPlanUniformPool& operator=(IFFTPlanUniformPool&& other)    = default;
74
75     /*=====
76     Member-Functions
77     -----*/
78     AccessPairs fetch_plan()
79     {
80         // setting the number of rounds to take
81         const int num_rounds {12};
82
83         // performing rounds
84         for(auto round = 0; round < num_rounds; ++round)
85         {
86             // going through vector mutexes
87             for(auto i =0; i < mutexes.size(); ++i)
88             {
89                 // trying to lock current mutex
90                 std::unique_lock<std::mutex> curr_lock(mutexes[i],
91                 std::try_to_lock);
92
93                 // if our lock contains the mutex, returning the plan and lock
94                 if (curr_lock.owns_lock())
95                     return AccessPairs(plans[i], std::move(curr_lock));
96             }
97         }
98
99         // throwing error
100         throw std::runtime_error("FILE: IFFTPlanPoolClass.hpp | CLASS:
101             IFFTPlanUniformPool | REPORT: COULDN'T FIND ANY AVAILABLE PLANS");
102     }
103 };
104 }

```

---

## A.7 FFT Plan Pool Handle

```

1  #pragma once
2
3  /*=====
4  Dependencies
5  -----*/
6  #include "FFTPlanPoolClass.hpp"
7
8  namespace svr
9  {
10     template <
11         typename sourceType,
12         typename destinationType,
13         typename = std::enable_if_t<
14             std::is_same_v< sourceType, double > &&
15             std::is_same_v< destinationType, std::complex<double> >
16         >
17     >

```

```

18 struct FFTPlanUniformPoolHandle
19 {
20     /*=====
21     Core Members
22     -----*/
23     svr::FFTPlanUniformPool<sourceType, destinationType> uniform_pool;
24     std::mutex                                         mutex;
25     std::size_t                                       num_plans;
26     std::size_t                                       nfft;
27
28     /*=====
29     Special Member-functions
30     -----*/
31     FFTPlanUniformPoolHandle() = default;
32     FFTPlanUniformPoolHandle(const std::size_t num_plans_arg,
33                             const std::size_t nfft_arg)
34         :   uniform_pool(num_plans_arg, nfft_arg),
35             num_plans(num_plans_arg),
36             nfft(nfft_arg) {}
37     FFTPlanUniformPoolHandle(const FFTPlanUniformPoolHandle& other) = delete;
38     FFTPlanUniformPoolHandle& operator=(const FFTPlanUniformPoolHandle& other) =
39         delete;
40     FFTPlanUniformPoolHandle(FFTPlanUniformPoolHandle&& other) = delete;
41     FFTPlanUniformPoolHandle& operator=(FFTPlanUniformPoolHandle&& other) = delete;
42
43     /*=====
44     Member Functions
45     -----*/
46     auto lock()
47     {
48         return std::unique_lock<std::mutex>(this->mutex);
49     }
50 };

```

---

## A.8 IFFT Plan Pool Handle

```

1  #pragma once
2
3  /*=====
4  Dependencies
5  -----*/
6  #include "IFFTPlanPoolClass.hpp"
7
8
9  namespace svr
10 {
11     template <
12         typename sourceType,
13         typename destinationType,
14         typename = std::enable_if_t<
15             std::is_same_v< sourceType,      std::complex<double> > &&
16             std::is_same_v< destinationType, double >
17         >
18     >
19     struct IFFTPlanUniformPoolHandle

```



```

20 {
21     /*=====
22     Members
23     -----*/
24     IFFTPlanUniformPool< sourceType,
25                          destinationType >    uniform_pool;
26     std::mutex                mutex;
27     std::size_t              num_plans;
28     std::size_t              nfft;
29
30     /*=====
31     Special Member Functions
32     -----*/
33     IFFTPlanUniformPoolHandle() = default;
34     IFFTPlanUniformPoolHandle(const std::size_t num_plans_arg,
35                               const std::size_t nfft_arg)
36     :    uniform_pool(    num_plans_arg, nfft_arg),
37         num_plans(    num_plans_arg),
38         nfft(    nfft_arg) {}
39     IFFTPlanUniformPoolHandle(const IFFTPlanUniformPoolHandle& other) = delete;
40     IFFTPlanUniformPoolHandle& operator=(const IFFTPlanUniformPoolHandle& other) =
41         delete;
42     IFFTPlanUniformPoolHandle(IFFTPlanUniformPoolHandle&& other) = delete;
43     IFFTPlanUniformPoolHandle& operator=(IFFTPlanUniformPoolHandle&& other) = delete;
44
45     /*=====
46     Member Functions
47     -----*/
48     auto    lock()
49     {
50         return std::unique_lock<std::mutex>(this->mutex);
51     }
52 };
53 }

```

---

# Appendix B

## General Purpose Templated Functions

### B.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 }

```

---

## B.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 }

```

---

## B.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 }

```

---

## B.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 }

```

---

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

```

```

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

```

```

245     }
246
247 }
248
249 /*=====
250 Long-signal Conv1D with FFT-Plan-Pool
251 -----*/
252 template <
253     typename T,
254     typename = std::enable_if_t<
255         std::is_same_v<T, double> ||
256         std::is_same_v<T, float>
257     >
258 >
259 auto conv_per_plan(
260     const int i,
261     const int& input_signal_block_size,
262     const int& filter_size,
263     const int& block_output_length,
264     const std::vector<T>& large_signal,
265     std::vector<T> signal_block_zero_padded,
266     svr::FFTPlanUniformPoolHandle<T, std::complex<T>>& fft_pool_handle,
267     svr::IFFTPlanUniformPoolHandle<std::complex<T>, T>& ifft_pool_handle,
268     const std::vector<std::complex<T>>& filter_FFT,
269     std::vector<T> fftw_output,
270     std::vector<T> conv_output,
271     std::vector<T>& output_vector,
272     std::mutex& output_vector_mutex,
273     const auto& signal_size
274 )
275 {
276
277     // calculating bounds
278     auto analytical_start {
279         (i*static_cast<int>(input_signal_block_size)) -
280         (static_cast<int>(filter_size) - 1)
281     };
282     auto analytical_end { (i+1)*input_signal_block_size - 1 };
283     auto start_index = std::max(
284         static_cast<int>(0), static_cast<int>(analytical_start)
285     );
286     auto end_index = std::min(
287         static_cast<int>(signal_size-1), static_cast<int>(analytical_end)
288     ); // [start-index, end-index]
289
290     // copying values
291     signal_block_zero_padded = std::move(std::vector<double>(block_output_length,
292         0.0));
293     std::copy(
294         large_signal.begin() + start_index,
295         large_signal.begin() + end_index + 1,
296         signal_block_zero_padded.begin() + start_index - analytical_start
297     );
298
299     // fetching an fft and IFFT plan
300     auto fph_lock {fft_pool_handle.lock()};
301     auto ifph_lock {ifft_pool_handle.lock()};
302     auto fft_pair {fft_pool_handle.uniform_pool.fetch_plan()};
303     auto ifft_pair {ifft_pool_handle.uniform_pool.fetch_plan()};

```

```

302
303 // performing ifft(fft(x) * filter-FFT)
304 fftw_output = ifft_pair.plan.ifft_l2_conservd(
305     fft_pair.plan.fft_l2_conservd(signal_block_zero_padded) * filter_FFT
306 );
307
308 // trimming away the first parts (since partial)
309 conv_output = std::vector<T>(
310     fftw_output.begin() + filter_size - 1,
311     fftw_output.end()
312 );
313
314 // writing to final-output
315 auto output_start_index = i * (block_output_length - (filter_size - 1));
316 std::lock_guard<std::mutex> output_lock(output_vector_mutex);
317 std::copy(
318     conv_output.begin(), conv_output.end(),
319     output_vector.begin() + output_start_index
320 );
321 }
322
323
324 template <
325     typename T,
326     typename = std::enable_if_t<
327         std::is_same_v<T, double> ||
328         std::is_same_v<T, float>
329     >
330 >
331 auto conv1D_long_FFTPlanPool(
332     const std::vector<T>& input_vector_A,
333     const std::vector<T>& input_vector_B,
334     svr::FFTPlanUniformPoolHandle<T, std::complex<T>>& fft_pool_handle,
335     svr::IFFTPlanUniformPoolHandle<std::complex<T>, T>& ifft_pool_handle
336 )
337 {
338     // Error checks
339     if (fft_pool_handle.nfft != ifft_pool_handle.nfft)
340         throw std::runtime_error("FILE: svr_conv.hpp | FUNCTION:
341             conv1D_long_FFTPlanPool | Report: the pool-handles are for different
342             nffts");
343
344     // fetching references to the large signal and small signal
345     const auto& large_signal_original {
346         input_vector_A.size() >= input_vector_B.size() ?
347         input_vector_A : input_vector_B
348     };
349     const auto& small_signal {
350         input_vector_A.size() < input_vector_B.size() ?
351         input_vector_A : input_vector_B
352     };
353
354     // copying
355     auto large_signal {std::vector<double>(
356         input_vector_A.size() + input_vector_B.size() - 1
357     )};
358     std::copy(large_signal_original.begin(),
359         large_signal_original.end(),
360         large_signal.begin());

```

```

359
360 // calculating some parameters
361 const auto signal_size {large_signal_original.size()};
362 const auto filter_size {small_signal.size()};
363 const auto input_signal_block_size {
364     fft_pool_handle.nfft + 1 - filter_size
365 };
366
367 // throwing an error if nfft < filter-size
368 if (fft_pool_handle.nfft < filter_size)
369     throw std::runtime_error("FILE: svr_conv.hpp | FUNCTION:
370         conv1D_long_FFTPlanPool | REPORT: filter is bigger than nfft");
371
372 // throwing an error if number of useful samples is less than zero
373 if (input_signal_block_size <= 0)
374     throw std::runtime_error("FILE: svr_conv.hpp | FUNCTION:
375         conv1D_long_FFTPlanPool | REPORT: input_signal_block_size = 0");
376
377 const auto block_output_length {fft_pool_handle.nfft};
378 const auto num_blocks {static_cast<int>((
379     1 + std::ceil((signal_size + filter_size - 2) / input_signal_block_size)
380 ));
381 const auto final_output_size {signal_size + filter_size - 1};
382 const auto useful_sample_length {
383     block_output_length - (filter_size - 1) - (filter_size - 1)
384 };
385
386 // parameters for re-use
387 auto start_index {static_cast<int>(0)};
388 auto end_index {static_cast<int>(0)};
389 auto output_start_index {static_cast<int>(0)};
390
391 // calculating fft(filter)
392 auto filter_zero_padded {std::vector<double>(block_output_length, 0.0)};
393 std::copy(small_signal.begin(), small_signal.end(), filter_zero_padded.begin());
394 auto fph_lock0 {fft_pool_handle.lock()};
395 auto curr_plan_pair {fft_pool_handle.uniform_pool.fetch_plan()};
396 auto pool_num_plans {fft_pool_handle.num_plans};
397 fph_lock0.unlock();
398 auto filter_FFT {
399     curr_plan_pair.plan.fft(
400         filter_zero_padded
401     )
402 };
403 curr_plan_pair.lock.unlock();
404
405 // allocating space for storing input-blocks
406 auto signal_block_zero_padded {std::vector<T>(block_output_length, 0.0)};
407 auto fftw_output {std::vector<T>()};
408 auto conv_output {std::vector<T>()};
409 auto output_vector {std::vector<T>(final_output_size, 0.0)};
410 auto output_vector_mutex {std::mutex()};
411
412 // creating boost
413 svr::ThreadPool local_pool(pool_num_plans);
414
415 // going through the values
416 for(auto i = 0; i < num_blocks; ++i)

```

```

416     {
417         local_pool.push_back(
418             [
419                 i,
420                 &input_signal_block_size,
421                 &filter_size,
422                 &block_output_length,
423                 &large_signal,
424                 signal_block_zero_padded,
425                 &fft_pool_handle,
426                 &ifft_pool_handle,
427                 &filter_FFT,
428                 fftw_output,
429                 conv_output,
430                 &output_vector,
431                 &output_vector_mutex,
432                 &signal_size
433             ]{
434                 conv_per_plan<T>(
435                     i,
436                     std::ref(input_signal_block_size),
437                     std::ref(filter_size),
438                     std::ref(block_output_length),
439                     std::ref(large_signal),
440                     signal_block_zero_padded,
441                     fft_pool_handle,
442                     ifft_pool_handle,
443                     filter_FFT,
444                     fftw_output,
445                     conv_output,
446                     std::ref(output_vector),
447                     output_vector_mutex,
448                     signal_size
449                 );
450             }
451         );
452     }
453     local_pool.converge();
454
455     // returning final output
456     // return std::move(output_vector);
457     return output_vector;
458 }
459
460 /*=====
461 Short-conv1D
462 -----*/
463 // template <std::size_t shortsize,
464 //          typename T1,
465 //          typename T2>
466 template <typename T1,
467          typename T2>
468 auto conv1D_short(const std::vector<T1>& input_vector_A,
469                  const std::vector<T2>& input_vector_B)
470 {
471     // resulting type
472     using T3 = decltype(std::declval<T1>() * std::declval<T2>());
473
474     // creating canvas

```

```

475     auto    canvas_length    {input_vector_A.size() + input_vector_B.size() - 1};
476
477     // calculating fft of two arrays
478     auto    fft_A            {svr::fft(input_vector_A, canvas_length)};
479     auto    fft_B            {svr::fft(input_vector_B, canvas_length)};
480
481     // element-wise multiplying the two matrices
482     auto    fft_AB           {fft_A *  fft_B};
483
484     // finding inverse FFT
485     auto    convolved_result {ifft(fft_AB)};
486
487     // returning
488     // return std::move(convolved_result);
489     return convolved_result;
490
491 }
492
493
494 /*=====
495 1D Convolution of a matrix and a vector
496 -----*/
497 template    <typename T>
498 auto    conv1D(const    std::vector<std::vector<T>>& input_matrix,
499                  const    std::vector<T>&                input_vector,
500                  const    std::size_t&                    dim)
501 {
502     // getting dimensions
503     const auto&    num_rows_matrix    {input_matrix.size()};
504     const auto&    num_cols_matrix    {input_matrix[0].size()};
505     const auto&    num_elements_vector {input_vector.size()};
506
507     // creating canvas
508     auto    canvas    {std::vector<std::vector<T>>()};
509
510     // creating output based on dim
511     if (dim == 1)
512     {
513         // performing convolutions row by row
514         for(auto    row = 0; row < num_rows_matrix; ++row)
515         {
516             cout << format("\t\t row = {}/{}\n", row, num_rows_matrix);
517             auto bruh {conv1D(input_matrix[row], input_vector)};
518             auto bruh_real {svr::real(std::move(bruh))};
519
520             canvas.push_back(
521                 svr::real(
522                     std::move(bruh_real)
523                 )
524             );
525         }
526     }
527     else{
528         std::cerr << "svr_conv.hpp | conv1D | yet to be implemented \n";
529     }
530
531     // returning
532     return std::move(canvas);
533

```

```

534     }
535
536     /*=====
537     f(Matrix, Vector)
538         - each row is convolved using the vector-argument
539     -----*/
540     template <
541         typename    firstType,
542         typename    secondType,
543         typename    =    std::enable_if_t<
544             std::is_same_v<firstType, std::complex<double>> &&
545             std::is_same_v<secondType, double>
546         >
547     >
548     auto    conv1D_PlanPool(
549         const    std::vector<std::vector<firstType>>&    input_matrix,
550         const    std::vector<secondType>&                input_vector,
551         svr::FFTPlanClass<firstType, firstType>&          fft_plan,
552         svr::IFFTPlanClass<firstType, firstType>&          ifft_plan
553     )
554     {
555         // error-checks
556         if (fft_plan.nfft_ != ifft_plan.nfft_)
557             throw    std::runtime_error(
558                 "FILE: svr_conv.hpp | FUNCTION: conv1D_PlanPool | REPORT: nfft-disparity
                    among plans"
559             );
560
561         // fetching dimensions
562         const    auto    num_rows_matrix    {input_matrix.size()};
563         const    auto    num_cols_matrix    {input_matrix[0].size()};
564         const    auto    num_rows_vector    {static_cast<std::size_t>(1)};
565         const    auto    num_cols_vector    {input_vector.size()};
566
567         // fetching references
568     }
569
570 }

```

---

## B.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";}

```

```

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 template <typename T>
113 auto sph2cart(const std::vector<T> spherical_vector){
114
115     // creating cartesian vector
116     auto cartesian_vector {std::vector<T>(spherical_vector.size(), 0)};
117
118     // populating
119     cartesian_vector[0] = spherical_vector[2] * \
120         cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
121         cos(spherical_vector[0] * std::numbers::pi / 180.00);
122     cartesian_vector[1] = spherical_vector[2] * \
123         cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
124         sin(spherical_vector[0] * std::numbers::pi / 180.00);
125     cartesian_vector[2] = spherical_vector[2] * \
126         sin(spherical_vector[1] * std::numbers::pi / 180.00);
127
128     // returning
129     return std::move(cartesian_vector);
130

```

```

131     }
132 }

```

---

## B.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 }

```

---

## B.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  };

```

```

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 };

```

---

## B.9 Editing Index Values

---

```

1  #pragma once
2  /*=====
3  Matlab's equivalent of A[A < 0.5] = 0
4  -----*/
5  template <typename T, typename U>
6  auto edit(std::vector<T>&          input_vector,
7           const std::vector<bool>& 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
26 /*=====
27 accumulate version of edit, instead of just placing values
28
29 Things to add
30 - ensuring template only accepts int, std::size_t and similar for T2
31 - bring in histogram method to ensure SIMD
32 -----*/
33 template <typename T1,
34          typename T2>
35 auto edit_accumulate(std::vector<T1>&          input_vector,
36                    const std::vector<T2>& indices_to_edit,
37                    const std::vector<T1>& new_values)
38 {
39     // certain checks
40     if (indices_to_edit.size() != new_values.size())
41         std::cerr << "svr::edit | edit_accumulate | size-disparity occurred \n";
42
43     // going through each and accumulating
44     for(auto i = 0; i < input_vector.size(); ++i){
45         const auto target_index {static_cast<std::size_t>(indices_to_edit[i])}; //

```

```

46     const auto new_value    {new_values[i]};
47     if (target_index < input_vector.size()){
48         input_vector[target_index] = input_vector[target_index] + new_value;
49     }
50     else{
51         // std::cout << "warning: FILE: svr_edit.hpp | FUNCTION: edit_accumulate |
52         // REPORT: index out of bounds";
53     }
54 }
55 // no-return since in-place
56 }

```

---

## B.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 }

```

---

## B.11 Exponentiate

```

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

```

```

17
18     // returning
19     return std::move(canvas);
20 }
21 /*=====
22 y = exp(matrix)
23 -----*/
24 template <
25     typename    sourceType,
26     typename    destinationType,
27     typename    =    std::enable_if_t<
28         std::is_arithmetic_v<sourceType>
29     >
30 >
31 auto    exp(
32     const    std::vector<std::vector<sourceType>>> input_matrix
33 )
34 {
35     // fetching dimensions
36     const    auto&    num_rows    {input_matrix.size()};
37     const    auto&    num_cols    {input_matrix[0].size()};
38
39     // creating canvas
40     auto    canvas    {std::vector<std::vector<destinationType>>>(
41         num_rows,
42         std::vector<destinationType>(num_cols)
43     )};
44
45     // writing to each entry
46     for(auto row = 0; row < num_rows; ++row)
47         std::transform(
48             input_matrix[row].begin(), input_matrix[row].end(),
49             canvas[row].begin(),
50             [](const    auto&    argx){
51                 return std::exp(argx);
52             }
53         );
54
55     // returning
56     return std::move(canvas);
57 }
58 /*=====
59 Aim: Exponentiating complex matrices with general floating types
60 -----*/
61 template <
62     typename    T,
63     typename    =    std::enable_if_t<
64         std::is_floating_point_v<T>
65     >
66 >
67 auto    exp(
68     const    std::vector<std::vector<std::complex<T>>>> input_matrix
69 )
70 {
71     // fetching dimensions
72     const    auto&    num_rows    {input_matrix.size()};
73     const    auto&    num_cols    {input_matrix[0].size()};
74
75     // creating canvas

```

```

76     auto    canvas    {std::vector<std::vector<std::complex<T>>>(
77         num_rows,
78         std::vector<std::complex<T>>(num_cols)
79     )});
80
81     // writing to each entry
82     for(auto row = 0; row < num_rows; ++row)
83         std::transform(
84             input_matrix[row].begin(), input_matrix[row].end(),
85             canvas[row].begin(),
86             [](const auto& argx){
87                 return std::exp(argx);
88             }
89         );
90
91     // returning
92     return std::move(canvas);
93 }
94
95 }

```

---

## B.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) {

```



```

92         in[i] = (i < input_vector.size()) ? input_vector[i] : 0.0;
93     }
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

```

```

150     for(int frequency_index = 0; frequency_index<nfft; ++frequency_index){
151         RType accumulate_value;
152         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                         static_cast<baseType>(signal_index))));
158         }
159         finaloutput[frequency_index] = accumulate_value / nfft_sqrt;
160     }
161
162     // returning
163     return std::move(finaloutput);
164 }
165
166 /*=====
167 x = ifft(std::vector<std::complex<double>> spectrum, nfft)
168 -----*/
169 #include <fftw3.h>
170 #include <vector>
171 #include <complex>
172 #include <stdexcept>
173
174 auto ifft(const std::vector<std::complex<double>>& input_vector,
175           const std::size_t nfft)
176 {
177     if (nfft <= 0)
178         throw std::runtime_error("nfft must be > 0");
179     if (input_vector.size() != nfft)
180         throw std::runtime_error("input spectrum must be of size nfft");
181
182     // Output: real-valued time-domain sequence
183     std::vector<double> output(nfft);
184
185     // Allocate FFTW input/output
186     fftw_complex* in = reinterpret_cast<fftw_complex*>(
187         fftw_malloc(sizeof(fftw_complex) * (nfft/2 + 1))
188     );
189     double* out = reinterpret_cast<double*>(
190         fftw_malloc(sizeof(double) * nfft)
191     );
192
193     // Copy *only* the first nfft/2+1 bins (rest are redundant due to symmetry)
194     for (std::size_t i = 0; i < nfft/2 + 1; ++i) {
195         in[i][0] = input_vector[i].real();
196         in[i][1] = input_vector[i].imag();
197     }
198
199     // Create inverse FFTW plan
200     fftw_plan plan = fftw_plan_dft_c2r_1d(
201         static_cast<int>(nfft),
202         in,
203         out,
204         FFTW_ESTIMATE
205     );
206
207     fftw_execute(plan);

```

```

208
209     // Normalize by nfft (FFTW leaves IFFT unscaled)
210     for (std::size_t i = 0; i < nfft; ++i) {
211         output[i] = out[i] / static_cast<double>(nfft);
212     }
213
214     // Cleanup
215     fftw_destroy_plan(plan);
216     fftw_free(in);
217     fftw_free(out);
218
219     return output;
220 }
221
222
223
224 }

```

---

## B.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 }

```

---

## B.14 Indexing

```

1  #pragma once
2  namespace svr {
3      /*=====
4      y = index(vector, mask)
5      =====
6      template <
7          typename T1,
8          typename T2,
9          typename = std::enable_if_t<
10              (std::is_arithmetic_v<T1> ||
11               std::is_same_v<T1, std::complex<float>> > ||
12               std::is_same_v<T1, std::complex<double>> >) &&
13              std::is_integral_v<T2>

```

```

14     >
15 >
16 -----*/
17 template <typename T1,
18           typename T2,
19           typename = std::enable_if_t<std::is_arithmetic_v<T1>          ||
20                                     std::is_same_v<T1, std::complex<float>> > ||
21                                     std::is_same_v<T1, std::complex<double>> >
22                                     >
23           >
24 auto index(const std::vector<T1>& input_vector,
25            const std::vector<T2>& indices_to_sample)
26 {
27     // creating canvas
28     auto canvas {std::vector<T1>(indices_to_sample.size(), 0)};
29
30     // copying the associated values
31     for(int i = 0; i < indices_to_sample.size(); ++i){
32         auto source_index {indices_to_sample[i]};
33         if(source_index < input_vector.size()){
34             canvas[i] = input_vector[source_index];
35         }
36         else{
37             // cout << "warning: Some chosen samples are out of bounds. svr::index |
38                 source_index !< input_vector.size()\n";
39
40         }
41     }
42
43     // returning
44     return std::move(canvas);
45 }
46 /*=====
47 y = index(matrix, mask, dim)
48 -----*/
49 template <
50     typename T1,
51     typename T2,
52     typename = std::enable_if_t<
53         (std::is_same_v<T1, double> || std::is_same_v<T1, float>) &&
54         (std::is_same_v<T2, int> || std::is_same_v<T2, std::size_t>)
55     >
56 >
57 auto index(const std::vector<std::vector<T1>>& input_matrix,
58            const std::vector<T2>& indices_to_sample,
59            const std::size_t& dim)
60 {
61     // fetching dimensions
62     const auto& num_rows_matrix {input_matrix.size()};
63     const auto& num_cols_matrix {input_matrix[0].size()};
64
65     // creating canvas
66     auto canvas {std::vector<std::vector<T1>>()};
67
68     // if indices are row-indices
69     if (dim == 0){
70
71         // initializing canvas
72         canvas = std::vector<std::vector<T1>>(<

```

```

72         num_rows_matrix,
73         std::vector<T1>(indices_to_sample.size())
74     );
75
76     // filling the canvas
77     auto destination_index {0};
78     std::for_each(
79         indices_to_sample.begin(), indices_to_sample.end(),
80         [&](const auto& col){
81             for(auto row = 0; row < num_rows_matrix; ++row){
82                 if (col <= input_matrix[0].size()){
83                     canvas[row][destination_index] = input_matrix[row][col];
84                 }
85             }
86             ++destination_index;
87         });
88 }
89 else if(dim == 1){
90     // initializing canvas
91     canvas = std::vector<std::vector<T1>>(
92         indices_to_sample.size(),
93         std::vector<T1>(num_cols_matrix)
94     );
95
96     // filling the canvas
97     #pragma omp parallel for
98     for(auto row = 0; row < canvas.size(); ++row){
99         auto destination_col {0};
100         std::for_each(indices_to_sample.begin(), indices_to_sample.end(),
101             [&row,
102              &input_matrix,
103              &destination_col,
104              &canvas](const auto& source_col){
105                 canvas[row][destination_col++] =
106                     input_matrix[row][source_col];
107             });
108     }
109     else {
110         std::cerr << "svr_index | this dim is not implemented \n";
111     }
112
113     // moving it back
114     return std::move(canvas);
115 }
116 }

```

---

## B.15 Linspace

```

1  /*=====
2  Dependencies
3  -----*/
4  #pragma once
5  #include <vector>
6  #include <complex>
7

```

```

8 namespace svr {
9  /*=====
10  in-place
11  -----*/
12  template <typename T>
13  auto linspace(
14      auto&          input,
15      const auto     startvalue,
16      const auto     endvalue,
17      const auto     numpoints
18  ) -> void
19  {
20      auto stepsize = static_cast<T>(endvalue - startvalue)/static_cast<T>(numpoints-1);
21      for(int i = 0; i<input.size(); ++i) {input[i] = startvalue + i*stepsize;}
22  };
23  /*=====
24  in-place
25  -----*/
26  template <typename T>
27  auto linspace(
28      std::vector<std::complex<T>>& input,
29      const auto                 startvalue,
30      const auto                 endvalue,
31      const auto                 numpoints
32  ) -> void
33  {
34      auto stepsize = static_cast<T>(endvalue - startvalue)/static_cast<T>(numpoints-1);
35      for(int i = 0; i<input.size(); ++i) {
36          input[i] = startvalue + static_cast<T>(i)*stepsize;
37      }
38  };
39  /*=====
40  -----*/
41  template <
42      typename T,
43      typename = std::enable_if_t<
44          std::is_arithmetic_v<T> ||
45          std::is_same_v<T, std::complex<float>> > ||
46          std::is_same_v<T, std::complex<double>> >
47      >
48  >
49  auto linspace(
50      const T          startvalue,
51      const T          endvalue,
52      const std::size_t numpoints
53  )
54  {
55      std::vector<T> input(numpoints);
56      auto stepsize = static_cast<T>(endvalue - startvalue)/static_cast<T>(numpoints-1);
57      for(int i = 0; i<input.size(); ++i) {input[i] = startvalue +
58          static_cast<T>(i)*stepsize;}
59      return std::move(input);
60  };
61  /*=====
62  -----*/
63  template <typename T, typename U>
64  auto linspace(
65      const T          startvalue,
66      const U          endvalue,

```

```

66     const    std::size_t    numpoints
67 )
68 {
69     std::vector<double> input(numpoints);
70     auto stepsize = static_cast<double>(endvalue -
71         startvalue)/static_cast<double>(numpoints-1);
72     for(int i = 0; i<input.size(); ++i) {input[i] = startvalue + i*stepsize;}
73     return std::move(input);
74 };
75 }

```

---

## B.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 }

```

---

## B.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 1-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 }

```

---

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

```



```

14
15     // returning the value
16     return std::move(canvas);
17 }

```

---

## B.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 */

```

---

## B.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 }
44 /*=====
45 element-wise with matrix
46 -----*/
47 template <
48     typename T,
49     typename = std::enable_if_t<
50         std::is_floating_point_v<T>
51     >

```

```

50 >
51 auto operator/((const std::vector<std::vector<T>>& input_matrix,
52                const T scalar)
53 {
54     // fetching matrix-dimensions
55     const auto& num_rows_matrix {input_matrix.size()};
56     const auto& num_cols_matrix {input_matrix[0].size()};
57
58     // creating canvas
59     auto canvas {std::vector<std::vector<T>>>(
60         num_rows_matrix,
61         std::vector<T>(num_cols_matrix)
62     )};
63
64     // dividing with values
65     for(auto row = 0; row < num_rows_matrix; ++row){
66         std::transform(input_matrix[row].begin(), input_matrix[row].end(),
67             canvas[row].begin(),
68             [&scalar](const auto& argx){
69                 return argx/scalar;
70             });
71     }
72
73     // returning values
74     return std::move(canvas);
75 }
76 template <
77     typename numeratorComplexType,
78     typename denominatorType,
79     typename = std::enable_if_t<
80         std::is_floating_point_v< numeratorComplexType> &&
81         std::is_arithmetic_v< denominatorType>
82     >
83 >
84 auto operator/((
85     const std::vector<std::vector<std::complex<numeratorComplexType>>>& input_matrix,
86     const denominatorType input_scalar
87 )
88 {
89     // fetching matrix-dimensions
90     const auto& num_rows_matrix {input_matrix.size()};
91     const auto& num_cols_matrix {input_matrix[0].size()};
92
93     // creating canvas
94     auto canvas {std::vector<std::vector<std::complex<numeratorComplexType>>>>(
95         num_rows_matrix,
96         std::vector<std::complex<numeratorComplexType>>>(num_cols_matrix)
97     )};
98
99     // dividing with values
100     for(auto row = 0; row < num_rows_matrix; ++row){
101         std::transform(
102             input_matrix[row].begin(), input_matrix[row].end(),
103             canvas[row].begin(),
104             [&input_scalar](const auto& argx){
105                 return argx /
106                     static_cast<std::complex<numeratorComplexType>>>(input_scalar);
107             });
108     }
109 }

```

```

108     // returning values
109     return std::move(canvas);
110 }
111
112 /*=====
113 y = std::vector<std::complex<T>> / T
114 -----*/
115 template <
116     typename T,
117     typename = std::enable_if_t<
118         std::is_floating_point_v<T>
119     >
120 >
121 auto operator/(
122     const std::vector<std::complex<T>>& input_vector,
123     const T input_scalar
124 )
125 {
126     // creating canvas
127     auto canvas {std::vector<std::complex<T>>(input_vector.size())};
128
129     // filling the canvas
130     std::transform(
131         input_vector.begin(), input_vector.end(),
132         canvas.begin(),
133         [&input_scalar](const auto& argx){
134             return argx/static_cast<std::complex<T>>(input_scalar);
135         }
136     );
137
138     // returning
139     return std::move(canvas);
140 }

```

---

## B.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 }

```

---

## B.22 Multiplication (Element-wise)

```

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

```



```

32 {
33     // fetching final-type
34     using T3 = decltype(std::declval<T1>() * std::declval<T2>());
35
36     // creating canvas
37     auto canvas {std::vector<T3>(input_vector.size())};
38
39     // multiplying
40     std::transform(
41         input_vector.begin(), input_vector.end(),
42         canvas.begin(),
43         [&scalar](auto& argx){
44             return static_cast<T3>(scalar) * static_cast<T3>(argx);
45         });
46
47     // returning
48     return std::move(canvas);
49 }
50 /*=====
51 y = scalar * vecotor (subset init)
52 -----*/
53 template <
54     typename T,
55     typename = std::enable_if_t<
56         std::is_floating_point_v<T>
57     >
58 >
59 auto operator*(
60     const std::complex<T> scalar,
61     const std::vector<T>& input_vector
62 )
63 {
64     // creating canvas
65     auto canvas {std::vector<std::complex<T>>(
66         input_vector.size()
67     )};
68
69     // copying to canvas
70     std::transform(
71         input_vector.begin(), input_vector.end(),
72         canvas.begin(),
73         [&scalar](const auto& argx){
74             return scalar * static_cast<std::complex<T>>(argx);
75         }
76     );
77
78     // moving it back
79     return std::move(canvas);
80 }
81 /*=====
82 y = vector * scalar
83 -----*/
84 template <typename T>
85 auto operator*(const std::vector<T>& input_vector,
86               const T scalar)
87 {
88     // creating canvas
89     auto canvas {input_vector};
90     // multiplying

```

```

91     std::for_each(canvas.begin(), canvas.end(),
92                   [&scalar](auto& argx){
93                         argx = argx * scalar;
94                   });
95     // returning
96     return std::move(canvas);
97 }
98 /*=====
99 y = vector * vector
100 -----*/
101 template <typename T>
102 auto operator*(const std::vector<T>& input_vector_A,
103               const std::vector<T>& input_vector_B)
104 {
105     // throwing error: size-disparity
106     if (input_vector_A.size() != input_vector_B.size()) {std::cerr << "operator*: size
107                                     disparity \n";}
108
109     // creating canvas
110     auto canvas = {input_vector_A};
111
112     // element-wise multiplying
113     std::transform(input_vector_B.begin(), input_vector_B.end(),
114                   canvas.begin(),
115                   canvas.begin(),
116                   [](const auto& argx, const auto& argy){
117                       return argx * argy;
118                   });
119
120     // moving it back
121     return std::move(canvas);
122 }
123 /*=====
124 y = vecotr * vector
125 -----*/
126 template <
127     typename T1,
128     typename T2,
129     typename = std::enable_if_t<
130         std::is_arithmetic_v<T1> &&
131         std::is_arithmetic_v<T2>
132     >
133 auto operator*(const std::vector<T1>& input_vector_A,
134               const std::vector<T2>& input_vector_B)
135 {
136
137     // checking size disparity
138     if (input_vector_A.size() != input_vector_B.size())
139         std::cerr << "operator*: error, size-disparity \n";
140
141     // figuring out resulting data type
142     using T3 = decltype(std::declval<T1>() * std::declval<T2>());
143
144     // creating canvas
145     auto canvas = {std::vector<T3>(input_vector_A.size())};
146
147     // performing multiplications
148     std::transform(input_vector_A.begin(), input_vector_A.end(),

```

```

149         input_vector_B.begin(),
150         canvas.begin(),
151         [](const auto& argx,
152            const auto& argy){
153             return static_cast<T3>(argx) * static_cast<T3>(argy);
154         });
155
156     // returning
157     return std::move(canvas);
158
159 }
160 /*=====
161 y = vector * complex_vector
162 -----*/
163 template <
164     typename T,
165     typename = std::enable_if_t<
166         std::is_floating_point_v<T>
167     >
168 >
169 auto operator*(
170     const std::vector<T>& input_vector_A,
171     const std::vector<std::complex<T>>& input_vector_B
172 )
173 {
174     // checking size issue
175     if (input_vector_A.size() != input_vector_B.size())
176         throw std::runtime_error(
177             "FILE: svr_operator_star.hpp | FUNCTION: operator* | REPORT: error disparity
178             in the two input-vectors"
179         );
180
181     // creating canvas
182     auto canvas {std::vector<std::complex<T>>( input_vector_A.size() )};
183
184     // filling up the canvas
185     std::transform(
186         input_vector_B.begin(), input_vector_B.end(),
187         input_vector_A.begin(),
188         canvas.begin(),
189         [](const auto& argx, const auto& argy){
190             return argx + static_cast<std::complex<T>>(argy);
191         });
192
193     // moving it back
194     return std::move(canvas);
195 }
196 /*=====
197 y = complex_vector * vector
198 -----*/
199 template <
200     typename T,
201     typename = std::enable_if_t<
202         std::is_floating_point_v<T>
203     >
204 >
205 auto operator*(
206     const std::vector<std::complex<T>>& input_vector_A,

```

```

207     const std::vector<T>&          input_vector_B
208 )
209 {
210     // enforcing size
211     if (input_vector_A.size() != input_vector_B.size())
212         throw std::runtime_error(
213             "FILE: svr_operator_star.hpp | FUNCTION: operator* overload"
214         );
215
216     // creating canvas
217     auto canvas {std::vector<std::complex<T>>(input_vector_A.size())};
218
219     // filling values
220     std::transform(
221         input_vector_A.begin(), input_vector_A.end(),
222         input_vector_B.begin(),
223         canvas.begin(),
224         [](const auto& argx, const auto& argy){
225             return argx * static_cast<std::complex<T>>(argy);
226         }
227     );
228
229     // returning
230     return std::move(canvas);
231 }
232 /*=====
233 y = complex-vector * complex-vector
234 -----*/
235 template <
236     typename T,
237     typename = std::enable_if_t<
238         std::is_floating_point_v<T>
239     >
240 >
241 auto operator*(
242     const std::vector<std::complex<T>> input_vector_A,
243     const std::vector<std::complex<T>> input_vector_B
244 )
245 {
246     // checking size
247     if (input_vector_A.size() != input_vector_B.size())
248         throw std::runtime_error(
249             "FILE: svr_operator_star.hpp | FUNCTION: operator*(complex-vector,
250                 complex-vector)"
251         );
252
253     // creating canvas
254     auto canvas {std::vector<std::complex<T>>(input_vector_A.size())};
255
256     // filling canvas
257     std::transform(
258         input_vector_A.begin(), input_vector_A.end(),
259         input_vector_B.begin(),
260         canvas.begin(),
261         [](const auto& argx, const auto& argy){
262             return argx * argy;
263         }
264     );

```

```

265     // returning values
266     return std::move(canvas);
267 }
268 // scalar * matrix =====
269 template <typename T>
270 auto operator*(const T scalar,
271               const std::vector<std::vector<T>>& inputMatrix)
272 {
273     std::vector<std::vector<T>> temp {inputMatrix};
274     for(int i = 0; i<inputMatrix.size(); ++i){
275         std::transform(inputMatrix[i].begin(),
276                       inputMatrix[i].end(),
277                       temp[i].begin(),
278                       [&scalar](T x){return scalar * x;});
279     }
280     return std::move(temp);
281 }
282 /*=====
283 y = matrix * scalar
284 -----*/
285 template <typename T>
286 auto operator*(const std::vector<std::vector<T>>& input_matrix,
287               const T scalar)
288 {
289     // fetching matrix dimensions
290     const auto& num_rows_matrix {input_matrix.size()};
291     const auto& num_cols_matrix {input_matrix[0].size()};
292
293     // creating canvas
294     auto canvas {std::vector<std::vector<T>>(
295         num_rows_matrix,
296         std::vector<T>(num_cols_matrix)
297     )};
298
299     // storing the values
300     for(auto row = 0; row < num_rows_matrix; ++row)
301         std::transform(input_matrix[row].begin(), input_matrix[row].end(),
302                       canvas[row].begin(),
303                       [&scalar](const auto& argx){
304                           return argx * scalar;
305                       });
306
307     // returning
308     return std::move(canvas);
309 }
310 /*=====
311 y = matrix * matrix
312 -----*/
313 template <typename T>
314 auto operator*(const std::vector<std::vector<T>>& A,
315               const std::vector<std::vector<T>>& B) -> std::vector<std::vector<T>>
316 {
317     // Case 1: element-wise multiplication
318     if (A.size() == B.size() && A[0].size() == B[0].size()) {
319         std::vector<std::vector<T>> C(A.size(), std::vector<T>(A[0].size()));
320         for (std::size_t row = 0; row < A.size(); ++row) {
321             std::transform(A[row].begin(), A[row].end(),
322                           B[row].begin(),
323                           C[row].begin(),

```

```

324         [](const auto& x, const auto& y){ return x * y; });
325     }
326     return C;
327 }
328
329 // Case 2: broadcast column vector
330 else if (A.size() == B.size() && B[0].size() == 1) {
331     std::vector<std::vector<T>> C(A.size(), std::vector<T>(A[0].size()));
332     for (std::size_t row = 0; row < A.size(); ++row) {
333         std::transform(A[row].begin(), A[row].end(),
334             C[row].begin(),
335             [&](const auto& x){ return x * B[row][0]; });
336     }
337     return C;
338 }
339
340 // case 3: when second matrix contains just one row
341 // case 4: when first matrix is just one column
342 // case 5: when second matrix is just one column
343
344 // Otherwise, invalid
345 else {
346     throw std::runtime_error("operator* dimension mismatch");
347 }
348 }
349
350 y = scalar * matrix
351 -----*/
352 template <typename T1, typename T2>
353 auto operator*(const T1 scalar,
354     const std::vector<std::vector<T2>>& inputMatrix)
355 {
356     std::vector<std::vector<T2>> temp {inputMatrix};
357     for(int i = 0; i<inputMatrix.size(); ++i){
358         std::transform(inputMatrix[i].begin(),
359             inputMatrix[i].end(),
360             temp[i].begin(),
361             [&scalar](T2 x){return static_cast<T2>(scalar) * x;});
362     }
363     return temp;
364 }
365
366 matrix-multiplication
367 -----*/
368 template <typename T1, typename T2>
369 auto matmul(const std::vector<std::vector<T1>>& matA,
370     const std::vector<std::vector<T2>>& matB)
371 {
372
373     // throwing error
374     if (matA[0].size() != matB.size()) {std::cerr << "dimension-mismatch \n";}
375
376     // getting result-type
377     using ResultType = decltype(std::declval<T1>() * std::declval<T2>() + \
378         std::declval<T1>() * std::declval<T2>());
379
380     // creating aliases
381     auto finalnumrows {matA.size()};
382     auto finalnumcols {matB[0].size()};

```

```

383
384 // creating placeholder
385 auto rowcolproduct = [&](auto rowA, auto colB){
386     ResultType temp {0};
387     for(int i = 0; i < matA.size(); ++i) {temp +=
388         static_cast<ResultType>(matA[rowA][i]) +
389         static_cast<ResultType>(matB[i][colB]);}
390     return temp;
391 };
392
393 // producing row-column combinations
394 std::vector<std::vector<ResultType>> finaloutput(finalnumrows,
395     std::vector<ResultType>(finalnumcols));
396 for(int row = 0; row < finalnumrows; ++row){for(int col = 0; col < finalnumcols;
397     ++col){finaloutput[row][col] = rowcolproduct(row, col);}}
398
399 // returning
400 return finaloutput;
401 }
402
403 /*=====
404 y = matrix * vector
405 -----*/
406
407 template <
408     typename T,
409     typename = std::enable_if_t<
410         std::is_arithmetic_v<T>
411     >
412 >
413 auto operator*(const std::vector<std::vector<T>>& input_matrix,
414     const std::vector<T>& input_vector)
415 {
416     // fetching dimensions
417     const auto& num_rows_matrix {input_matrix.size()};
418     const auto& num_cols_matrix {input_matrix[0].size()};
419     const auto& num_rows_vector {1};
420     const auto& num_cols_vector {input_vector.size()};
421     const auto& max_num_rows {num_rows_matrix > num_rows_vector ?\
422         num_rows_matrix : num_rows_vector};
423     const auto& max_num_cols {num_cols_matrix > num_cols_vector ?\
424         num_cols_matrix : num_cols_vector};
425
426     // creating canvas
427     auto canvas {std::vector<std::vector<T>>(
428         max_num_rows,
429         std::vector<T>(max_num_cols, 0)
430     )};
431
432     // multiplying column matrix with row matrix
433     if (num_cols_matrix == 1 && num_rows_vector == 1){
434
435         // writing to canvas
436         for(auto row = 0; row < max_num_rows; ++row)
437             for(auto col = 0; col < max_num_cols; ++col)
438                 canvas[row][col] = input_matrix[row][0] * input_vector[col];
439     }
440     /*=====
441     Multiplying each row with the input-vector
442     -----*/
443     else if(

```

```

438     num_cols_matrix == num_cols_vector &&
439     num_rows_vector == 1
440 )
441 {
442     // writing to canvas
443     for(auto row = 0; row < max_num_rows; ++row)
444         std::transform(
445             input_matrix[row].begin(), input_matrix[row].end(),
446             input_vector.begin(),
447             canvas[row].begin(),
448             [](const auto& argx, const auto& argy){return argx * argy;}
449         );
450 }
451 else
452 {
453     std::cerr << "Operator*: [matrix, vector] | not implemented \n";
454 }
455
456 // returning
457 return std::move(canvas);
458
459 }
460 /*=====
461 complex-matrix * vector
462 -----*/
463 template <
464     typename T,
465     typename = std::enable_if_t<
466         std::is_floating_point_v<T>
467     >
468 >
469 auto operator*(
470     const std::vector<std::vector<std::complex<T>>>& input_matrix,
471     const std::vector<T>& input_vector
472 )
473 {
474     // fetching dimensions
475     const auto num_rows_matrix {input_matrix.size()};
476     const auto num_cols_matrix {input_matrix[0].size()};
477     const auto num_rows_vector {static_cast<std::size_t>(1)};
478     const auto num_cols_vector {input_vector.size()};
479
480     // throwing an error
481     if (num_cols_matrix != num_cols_vector)
482         throw std::runtime_error(
483             "FILE: svr_operator_star.hpp | FUNCTION: operator*(complex-matrix, vector)"
484         );
485
486     // creating canvas
487     auto canvas {std::vector<std::vector<std::complex<T>>>(
488         num_rows_matrix,
489         std::vector<std::complex<T>>(num_cols_matrix)
490     )};
491
492     // performing operations
493     for(auto row = 0; row < num_rows_matrix; ++row)
494         std::transform(
495             input_matrix[row].begin(), input_matrix[row].end(),
496             input_vector.begin(),

```



```

497         canvas[row].begin(),
498         [](const auto& argx, const auto& argy){
499             return argx * static_cast<std::complex<T>>(argy);
500         }
501     );
502
503     // returning the final output
504     return std::move(canvas);
505
506 }
507 /*=====
508 martix * complex-vector
509 -----*/
510 template <
511     typename T,
512     typename = std::enable_if_t<
513         std::is_floating_point_v<T>
514     >
515 >
516 auto operator*(
517     const std::vector<std::vector<T>>& input_matrix,
518     const std::vector<std::complex<T>>& input_vector
519 )
520 {
521     // fetching dimensions
522     const auto num_rows_matrix {input_matrix.size()};
523     const auto num_cols_matrix {input_matrix[0].size()};
524     const auto num_rows_vector {static_cast<std::size_t>(1)};
525     const auto num_cols_vector {input_vector.size()};
526
527     // fetching dimension mismatch
528     if (num_cols_matrix != num_cols_vector)
529         throw std::runtime_error(
530             "FILE: svr_operator_star.hpp | FUNCTION: operator*(input-matrix,
531                 complex-vector)"
532         );
533
534     // creating canvas
535     auto canvas {std::vector<std::vector<std::complex<T>>>(
536         num_rows_matrix,
537         std::vector<std::complex<T>>(
538             num_cols_matrix
539         )
540     )};
541
542     // filling the values
543     for(auto row = 0; row < num_rows_matrix; ++row)
544         std::transform(
545             input_matrix[row].begin(), input_matrix[row].end(),
546             input_vector.begin(),
547             canvas[row].begin(),
548             [](const auto& argx, const auto& argy){
549                 return static_cast<std::complex<T>>(argx) * argy;
550             }
551         );
552
553     // returning final-output
554     return std::move(canvas);
555 }

```

```

555  /*=====
556  scalar operators
557  -----*/
558  auto operator*(const std::complex<double> complexscalar,
559                const double          doublescalar){
560      return complexscalar * static_cast<std::complex<double>>(doublescalar);
561  }
562  auto operator*(const double          doublescalar,
563                const std::complex<double> complexscalar){
564      return complexscalar * static_cast<std::complex<double>>(doublescalar);
565  }
566  auto operator*(const std::complex<double> complexscalar,
567                const int          scalar){
568      return complexscalar * static_cast<std::complex<double>>(scalar);
569  }
570  auto operator*(const int          scalar,
571                const std::complex<double> complexscalar){
572      return complexscalar * static_cast<std::complex<double>>(scalar);
573  }

```

---

## B.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 }

## B.24 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 -----*/
60 void fPrintLinkedList(ListNode* root){
61     if (root == nullptr) return;
62     cout << root->val << " -> ";
63     fPrintLinkedList(root->next);
64     return;
65 }
66 /*=====
67 -----*/
68 template<typename T>
69 void fPrintContainer(T input){
70     for(auto x: input) cout << x << ", ";
71     cout << endl;
72     return;
73 }
74 /*=====
75 -----*/
76 template <typename T>
77 auto size(std::vector<std::vector<T>> inputMatrix){
78     cout << format("[{}, {}]\n",
79                     inputMatrix.size(),
80                     inputMatrix[0].size());
81 }
82 /*=====
83 -----*/
84 template <typename T>
85 auto size(const std::string& inputstring,
86           const std::vector<std::vector<T>>& inputMatrix){
87     cout << format("{} = [{}, {}]\n",
88                     inputstring,
89                     inputMatrix.size(),
90                     inputMatrix[0].size());
91 }

```

---

## B.25 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 }

```

---

## B.26 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 }

```

---

## B.27 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,
6      std::vector<T>>
7  {
8      // returning the input as is
9      return input_vector;
10 }
11 /*=====
12 -----*/
13 template <std::size_t axis, typename T>
14 auto sum(const std::vector<std::vector<T>>& input_matrix) -> std::enable_if_t<axis == 0,
15     std::vector<T>>
16 {

```

```

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,
33         std::vector<std::vector<T>>>
34 {
35     // creating canvas
36     auto canvas {std::vector<std::vector<T>>(input_matrix.size(),
37                                             std::vector<T>(1, 0.00))};
38
39     // filling up the canvas
40     for(auto row = 0; row < input_matrix.size(); ++row)
41         canvas[row][0] = std::accumulate(input_matrix[row].begin(),
42                                           input_matrix[row].end(),
43                                           static_cast<T>(0));
44
45     // returning
46     return std::move(canvas);
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,
53         std::vector<T> >
54 {
55     // setup
56     const auto& num_cols_A {input_vector_A.size()};
57     const auto& num_cols_B {input_vector_B.size()};
58
59     // throwing errors
60     if (num_cols_A != num_cols_B) {std::cerr << "sum: size disparity\n";}
61
62     // creating canvas
63     auto canvas {input_vector_A};
64
65     // summing up
66     std::transform(input_vector_B.begin(), input_vector_B.end(),
67                   canvas.begin(),
68                   canvas.begin(),
69                   std::plus<T>());
70
71     // returning
72     return std::move(canvas);

```

72 }

## B.28 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 }
```

---

## B.29 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 /*=====
35 tiling a matrix -----*/
36
37 template <typename T>
38 auto tile(const std::vector<std::vector<T>>& input_matrix,
39     const std::vector<size_t>& mul_dimensions){
40
41     // creating canvas
42     const std::size_t& num_rows {input_matrix.size() * mul_dimensions[0]};
43     const std::size_t& num_cols {input_matrix[0].size() * mul_dimensions[1]};
44     auto canvas {std::vector<std::vector<T>>>(
45         num_rows,
46         std::vector<T>(num_cols, 0)
47     )};
48
49     // writing
50     std::size_t source_row;
51     std::size_t source_col;
52
53     for(std::size_t row = 0; row < num_rows; ++row){
54         for(std::size_t col = 0; col < num_cols; ++col){
55             source_row = row % input_matrix.size();
56             source_col = col % input_matrix[0].size();
57             canvas[row][col] = input_matrix[source_row][source_col];
58         }
59     }
60
61     // returning
62     return std::move(canvas);
63 }

```

---

## B.30 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 }

```

---

## B.31 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 canvas
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 }

```

---

## B.32 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 }

```

---

## B.33 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        // 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 }

```

---

## B.34 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 }

## B.35 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 }
```

---

## B.36 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 }

```

---

## B.37 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 }

```

---

## B.38 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 }

```

---