Autonomous Underwater Vehicle: A Surveillance Protocol

S.V. Rajendran

October 18, 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

Part I

AUV Components & Setup

Chapter 1

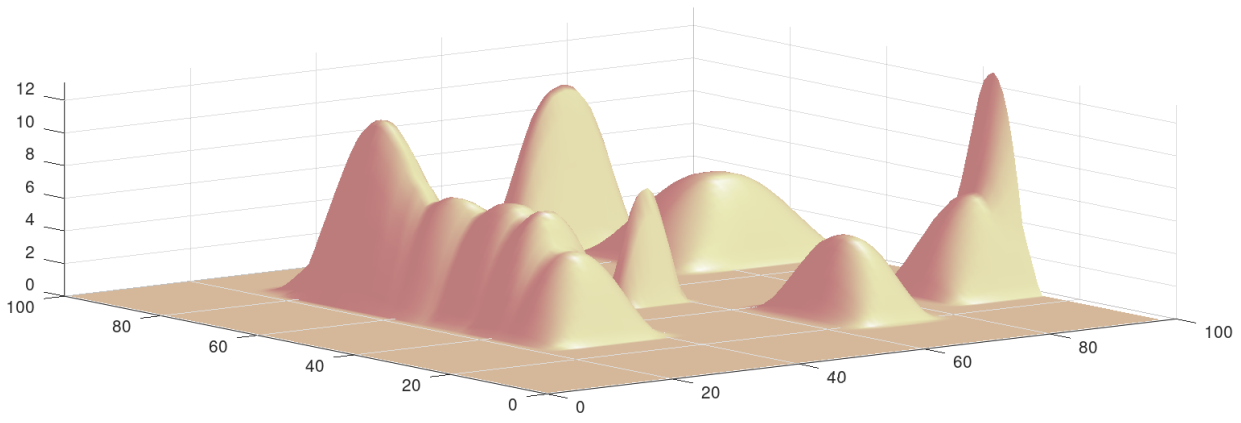
Underwater Environment

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

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

1.2 Scatterer Definition

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

```
/*================================================================================
Class Declaration
--------------------------------------------------------------------------------*/
template <typename T>
class ScattererClass
{
public:
    // members
    std::vector<std::vector<T>> coordinates;
    std::vector<T>              reflectivity;

    // Constructor
    ScattererClass()  {}

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

    // Save to CSV
```

```
    void save_to_csv();
};
```

1.3 Sea-Floor Setup Script

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

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

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

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

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

    // scatter density
    // auto bed_width_density {static_cast<double>( 10.00)};
    // auto   bed_length_density {static_cast<double>( 10.00)};
    auto   bed_width_density {static_cast<double>( 5.00)};
    auto   bed_length_density {static_cast<double>( 5.00)};

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

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

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

    // creating heights of scatterers
    if(hill_creation_flag){
```

```
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 -
                                               hill_dimensions_min)};              // verified
66
67          // function-call: hill-creation function
68          fCreateHills(hill_2D_center,
69                       hill_dimensions,
70                       points2D);
71
72          // setting up floor reflectivity
73          auto   floorScatter_reflectivity     {std::vector<double>(Y.size(), 1.00)};
74
75          // populating the values of the incoming argument
76          scatterers.coordinates    = std::move(points2D);
77          scatterers.reflectivity   = std::move(floorScatter_reflectivity);
78
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      spdlog::info("Finished Sea-Floor Setup");
97  }
```

Chapter 2

Transmitter

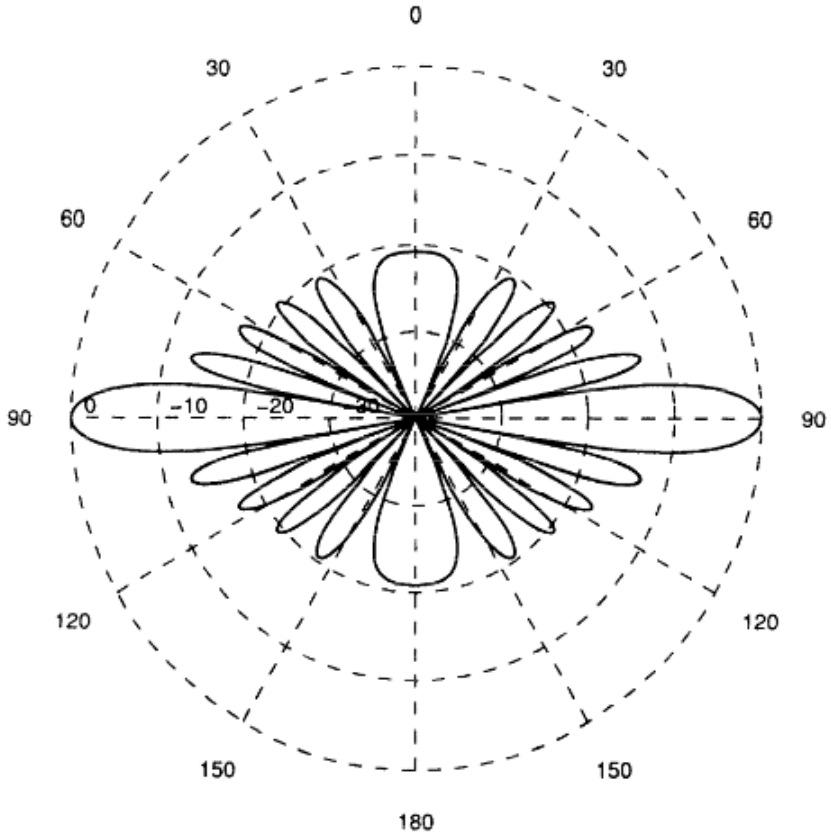


Figure 2.1: Beampattern of a Transmission Uniform Linear Array

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

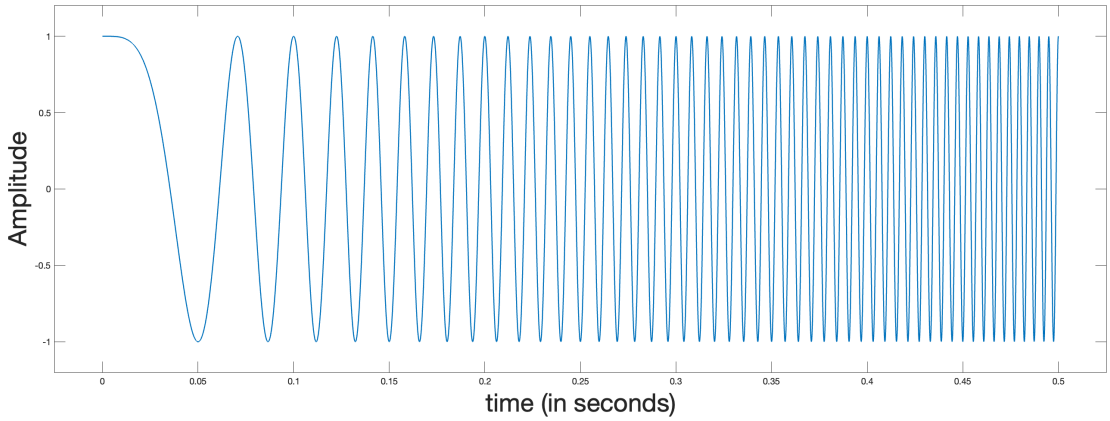


Figure 2.2: Linear Frequency Modulated Wave

The resolution of any probing system is fundamentally tied to the signal bandwidth. A higher bandwidth corresponds to finer resolution $\frac{\text{speed-of-sounds}}{2*\text{bandwidth}}$. 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
           azimuth
29     int               elevationQuantDensity;   // quantization of angles along the
           elevation
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
                                = 0
31
32      // Setting up transmitter
33      auto    location                        {std::vector<T>(3, 0)};     // location of
            transmitter
34      T       azimuthal_angle_fls             {0};                        // initial
            pointing direction
35      T       azimuthal_angle_port            {90};                       // initial
            pointing direction
36      T       azimuthal_angle_starboard       {-90};                      // initial
            pointing direction
37
38      T       elevation_angle                 {-60};                      // initial
            pointing direction
39
40      T       azimuthal_beamwidth_fls         {20};                       // azimuthal
            beamwidth of the signal cone
41      T       azimuthal_beamwidth_port        {20};                       // azimuthal
            beamwidth of the signal cone
42      T       azimuthal_beamwidth_starboard   {20};                       // azimuthal
            beamwidth of the signal cone
43
44      T       elevation_beamwidth_fls         {20};                       // elevation
            beamwidth of the signal cone
45      T       elevation_beamwidth_port        {20};                       // elevation
            beamwidth of the signal cone
46      T       elevation_beamwidth_starboard   {20};                       // elevation
            beamwidth of the signal cone
47
48      int     azimuthQuantDensity             {10};  // number of points, a degree is split
            into quantization density along azimuth (used for shadowing)
49      int     elevationQuantDensity           {10};  // number of points, a degree is split
            into quantization density along elevation (used for shadowing)
50      T       rangeQuantSize                  {10};  // the length of a cell (used for
            shadowing)
51
52      T       azimuthShadowThreshold          {1};   // azimuth threshold   (in degrees)
53      T       elevationShadowThreshold        {1};   // elevation threshold (in degrees)
54
55
56      // transmitter-fls
57      transmitter_fls.location            = location;                     // Assigning
            location
58      transmitter_fls.signal              = Signal;                       // Assigning
            signal
59      transmitter_fls.azimuthal_angle     = azimuthal_angle_fls;          // assigning
            azimuth angle
60      transmitter_fls.elevation_angle     = elevation_angle;              // assigning
            elevation angle
61      transmitter_fls.azimuthal_beamwidth = azimuthal_beamwidth_fls;      // assigning
            azimuth-beamwidth
62      transmitter_fls.elevation_beamwidth = elevation_beamwidth_fls;      // assigning
            elevation-beamwidth
63      // updating quantization densities
64      transmitter_fls.azimuthQuantDensity    = azimuthQuantDensity;     // assigning
            azimuth quant density
65      transmitter_fls.elevationQuantDensity  = elevationQuantDensity;   // assigning
            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

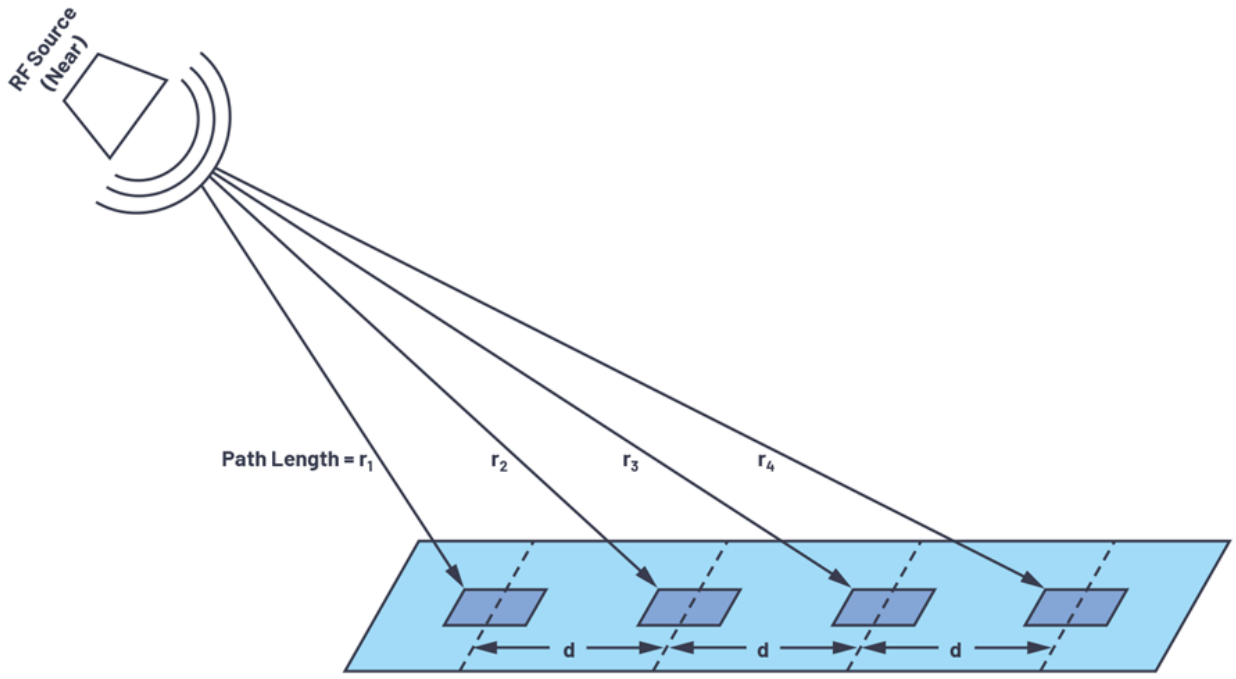


Figure 3.1: Uniform Linear Array

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

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

    // derived
    std::vector<T>              sensor_direction;
    std::vector<std::vector<T>> signal_matrix;
    std::size_t                 num_samples;

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

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

```
32      int  frame_size;                // the frame-size corresponding to a range cell in a
            decimated signal matrix
33
34      std::vector<std::vector<complex<T>>> mulFFTMatrix; // the matrix containing the
            delays for each-element as a slot
35      std::vector<complex<T>>             matchFilter;  // torch tensor containing the
            match-filter
36      int   num_buffer_zeros_per_frame;                 // number of zeros we're adding
            per frame to ensure no-rotation
37      std::vector<std::vector<T>>  beamformedImage;      // the beamformed image
38      std::vector<std::vector<T>>  cartesianImage;       // the cartesian version of
            beamformed image
39
40      // Decimating Related
41      std::vector<std::complex<T>> basebanding_signal;
42
43      // Artificial acoustic-image related
44      std::vector<std::vector<T>>  currentArtificialAcousticImage; // acoustic image
            directly produced
45
46
47      // Basic Constructor
48      ULAClass() = default;
49
50      // constructor
51      ULAClass(const int    num_sensors_arg,
52               const auto   inter_element_spacing_arg,
53               const auto&  coordinates_arg,
54               const auto&  sampling_frequency_arg,
55               const auto&  recording_period_arg,
56               const auto&  location_arg,
57               const auto&  signalMatrix_arg,
58               const auto&  lowpass_filter_coefficients_for_decimation_arg):
59                  num_sensors(num_sensors_arg),
60                  inter_element_spacing(inter_element_spacing_arg),
61                  coordinates(std::move(coordinates_arg)),
62                  sampling_frequency(sampling_frequency_arg),
63                  recording_period(recording_period_arg),
64                  location(std::move(location_arg)),
65                  signal_matrix(std::move(signalMatrix_arg)),
66                  lowpass_filter_coefficients_for_decimation(std::move(lowpass_filter_coefficients_for_decima
67      {
68
69          // calculating ULA direction
70          sensor_direction = std::vector<T>{coordinates[1][0] - coordinates[0][0],
71                                            coordinates[1][1] - coordinates[0][1],
72                                            coordinates[1][2] - coordinates[0][2]};
73
74          // normalizing
75          auto    norm_value_temp   {std::norm(std::inner_product(sensor_direction.begin(),
76                                                                  sensor_direction.end(),
77                                                                  sensor_direction.begin(),
78                                                                  0.00))};
79
80          // dividing
81          if (norm_value_temp != 0) {sensor_direction =  sensor_direction /
                norm_value_temp;}
82
83      }
```

```
84
85      // // deleting copy constructor/assignment
86      // ULAClass<T>(const ULAClass<T>& other)                 = delete;
87      // ULAClass<T>& operator=(const ULAClass<T>& other)      = delete;
88      ULAClass<T>(ULAClass<T>&& other)                      = delete;
89      ULAClass<T>&   operator=(const ULAClass<T>& other)    = default;
90
91      // member-functions
```

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>(32)};         // number of sensors
15      T       sampling_frequency      {static_cast<T>(160e3)};        // sampling frequency
16      T       inter_element_spacing   {1500/(2*sampling_frequency)};  // space between
            samples
17      T       recording_period        {10e-2};                        // sampling-period
18      auto    num_samples             {static_cast<std::size_t>(
19          std::ceil(
20              sampling_frequency  *   recording_period
21          )
22      )};
23
24      // building the direction for the sensors
25      auto    ULA_direction           {std::vector<T>({-1, 0, 0})};
26      auto    ULA_direction_norm      {norm(ULA_direction)};
27      if (ULA_direction_norm != 0)    {ULA_direction = ULA_direction/ULA_direction_norm;}
28      ULA_direction        =          ULA_direction   *   inter_element_spacing;
29
30      // building coordinates for sensors
31      auto    ULA_coordinates         {
32          transpose(ULA_direction) * \
33          svr::linspace<double>(
34              0.00,
35              num_sensors -1,
36              num_sensors)
37      };
38
39      // coefficients of decimation filter
40      auto    lowpassfiltercoefficients {std::vector<T>{0.0000, 0.0000, 0.0000, 0.0000,
            0.0000, 0.0000, 0.0001, 0.0003, 0.0006, 0.0015, 0.0030, 0.0057, 0.0100, 0.0163,
            0.0251, 0.0364, 0.0501, 0.0654, 0.0814, 0.0966, 0.1093, 0.1180, 0.1212, 0.1179,
```

```
        0.1078, 0.0914, 0.0699, 0.0451, 0.0192, -0.0053, -0.0262, -0.0416, -0.0504,
        -0.0522, -0.0475, -0.0375, -0.0239, -0.0088, 0.0057, 0.0179, 0.0263, 0.0303,
        0.0298, 0.0253, 0.0177, 0.0086, -0.0008, -0.0091, -0.0153, -0.0187, -0.0191,
        -0.0168, -0.0123, -0.0065, -0.0004, 0.0052, 0.0095, 0.0119, 0.0125, 0.0112,
        0.0084, 0.0046, 0.0006, -0.0031, -0.0060, -0.0078, -0.0082, -0.0075, -0.0057,
        -0.0033, -0.0006, 0.0019, 0.0039, 0.0051, 0.0055, 0.0050, 0.0039, 0.0023, 0.0005,
        -0.0012, -0.0025, -0.0034, -0.0036, -0.0034, -0.0026, -0.0016, -0.0004, 0.0007,
        0.0016, 0.0022, 0.0024, 0.0023, 0.0018, 0.0011, 0.0003, -0.0004, -0.0011,
        -0.0015, -0.0016, -0.0015}};

    // assigning values
    ula_fls.num_sensors                             = num_sensors;
    ula_fls.inter_element_spacing                   = inter_element_spacing;
    ula_fls.coordinates                             = ULA_coordinates;
    ula_fls.sampling_frequency                      = sampling_frequency;
    ula_fls.recording_period                        = recording_period;
    ula_fls.sensor_direction                        = ULA_direction;
    ula_fls.lowpass_filter_coefficients_for_decimation = lowpassfiltercoefficients;
    ula_fls.num_samples                             = num_samples;


    // assigning values
    ula_portside.num_sensors                                = num_sensors;
    ula_portside.inter_element_spacing                      = inter_element_spacing;
    ula_portside.coordinates                                = ULA_coordinates;
    ula_portside.sampling_frequency                         = sampling_frequency;
    ula_portside.recording_period                           = recording_period;
    ula_portside.sensor_direction                           = ULA_direction;
    ula_portside.lowpass_filter_coefficients_for_decimation = lowpassfiltercoefficients;
    ula_portside.num_samples                                =   num_samples;


    // assigning values
    ula_starboard.num_sensors                               = num_sensors;
    ula_starboard.inter_element_spacing                     = inter_element_spacing;
    ula_starboard.coordinates                               = ULA_coordinates;
    ula_starboard.sampling_frequency                        = sampling_frequency;
    ula_starboard.recording_period                          = recording_period;
    ula_starboard.sensor_direction                          = ULA_direction;
    ula_starboard.lowpass_filter_coefficients_for_decimation = lowpassfiltercoefficients;
    ula_starboard.num_samples                               = num_samples;
}
```

Chapter 4

Autonomous Underwater Vehicle

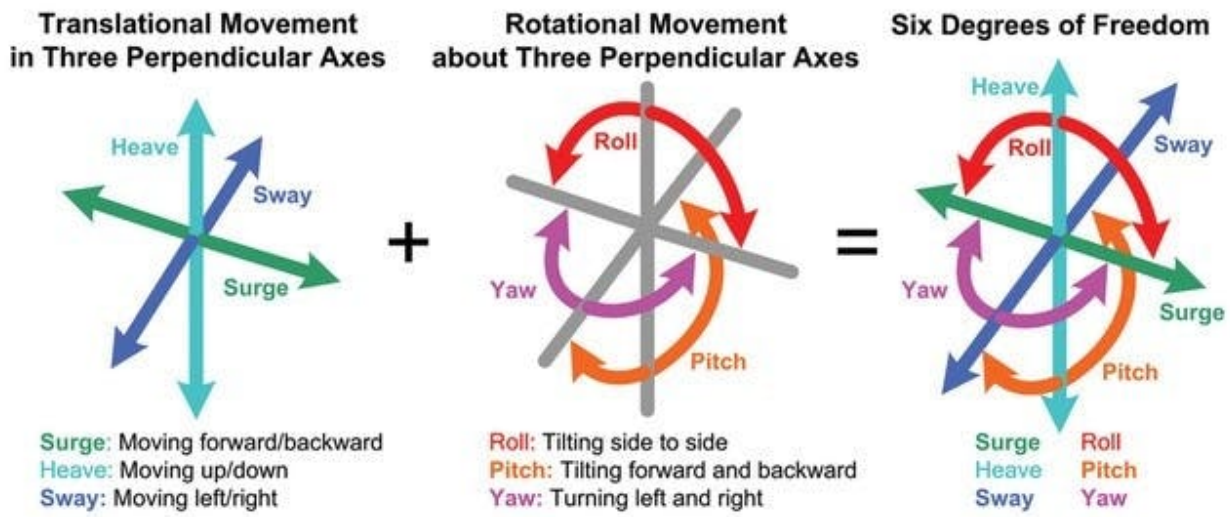


Figure 4.1: AUV degrees of freedom

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

```
template <
    svr::PureFloatingPointType      T,
    svr::FFT_SourceDestination_Type sourceType,
    svr::FFT_SourceDestination_Type destinationType
>
class  AUVClass{
public:

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

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

    // transmitters
    TransmitterClass<T>   transmitter_fls;        // transmitter for front-looking SONAR
    TransmitterClass<T>   transmitter_portside;   // portside transmitter
    TransmitterClass<T>   transmitter_starboard;  // starboard transmitter

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

    // plotting mode
```

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

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

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

    // functions
    void   syncComponentAttributes();
    void   init(
        svr::ThreadPool&                                                thread_pool,
        svr::FFTPlanUniformPoolHandle<T,           std::complex<T>>& fph_match_filter,
        svr::IFFTPlanUniformPoolHandle<std::complex<T>, T>&          ifph_match_filter);
    void   simulate_signal(
        const    ScattererClass<T>&                       seafloor,
        svr::ThreadPool&                                  thread_pool,
        svr::FFTPlanUniformPoolHandle<T, std::complex<T>>& fft_pool_handle,
        svr::IFFTPlanUniformPoolHandle<std::complex<T>, T>& ifft_pool_handle);
    void   subset_scatterers(
        const ScattererClass<T>&     seafloor,
        svr::ThreadPool&             thread_pool,
```

4.2 AUV Setup Scripts

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

```
template <
    svr::PureFloatingPointType      T,
    svr::FFT_SourceDestination_Type  sourceType,
    svr::FFT_SourceDestination_Type  destinationType
>
void fAUVSetup(
    AUVClass<T, sourceType, destinationType>& auv
) {

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

    // assigning
    auv.location          = std::move(location);          // assigning location
    auv.velocity          = std::move(velocity);          // assigning velocity
    auv.pointing_direction = std::move(pointing_direction); // assigning pointing
        direction

    // signaling end
    spdlog::info("Completed AUV-setup");
}
```

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 does'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

```
#pragma once
/*============================================================================
writing the contents of a vector a csv-file
----------------------------------------------------------------------------*/
template <typename T>
void fWriteVector(const vector<T>&         inputvector,
                  const string&            filename){

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

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

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

    // return
    return;
}
/*============================================================================
```

```
writing the contents of a matrix to a csv-file
-------------------------------------------------------------------------------*/
template <typename T>
auto fWriteMatrix(const std::vector<std::vector<T>> inputMatrix,
                  const string                      filename){

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

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

}
/*===============================================================================
writing complex-matrix to a csv-file
-------------------------------------------------------------------------------*/
template <>
auto fWriteMatrix(const std::vector<std::vector<std::complex<double>>> inputMatrix,
                  const string                                         filename){

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

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

A.2 Thread-Pool

```
#pragma once
namespace  svr {
    class ThreadPool {
    public:
```

```
        // Members
        boost::asio::thread_pool     thread_pool;      // the pool
        std::vector<std::future<void>> future_vector;  // futures to wait on

        // Special-Members
        ThreadPool(std::size_t num_threads) : thread_pool(num_threads) {}
        ThreadPool(const ThreadPool& other)     = delete;
        ThreadPool& operator=(ThreadPool& other) = delete;

        // Member-functions
        void                        converge();
        template <typename F> void push_back(F&& func);
        void                        shutdown();

    private:
        template<typename F>
        std::future<void> _wrap_task(F&& func) {
            std::promise<void> p;
            auto f = p.get_future();

            boost::asio::post(thread_pool,
                [func = std::forward<F>(func), p = std::move(p)]() mutable {
                    func();
                    p.set_value();
                });

            return f;
        }
    };

    /*============================================================================
    Member-Function: Add new task to the pool
    ----------------------------------------------------------------------------*/
    template   <typename F>
    void ThreadPool::push_back(F&& func)
    {
        future_vector.push_back(_wrap_task(std::forward<F>(func)));
    }
    /*============================================================================
    Member-Function: waiting until all the assigned work is done
    ----------------------------------------------------------------------------*/
    void ThreadPool::converge()
    {
        for (auto &fut : future_vector) fut.get();
        future_vector.clear();
    }
    /*============================================================================
    Member-Function: Shutting things down
    ----------------------------------------------------------------------------*/
    void ThreadPool::shutdown()
    {
        thread_pool.join();
    }

}
```

A.3 FFTPlanClass

```
#pragma once

namespace  svr    {

    template   <typename T>
    concept FFTPlanClassSourceDestinationType = \
        std::is_floating_point_v<T> ||
        (
            std::is_class_v<T> &&
            std::is_floating_point_v<typename T::value_type>
        );
    template   <
        FFTPlanClassSourceDestinationType sourceType,
        FFTPlanClassSourceDestinationType destinationType
    >
    class FFTPlanClass
    {
        public:

            // Members
            std::size_t    nfft_      {std::numeric_limits<std::size_t>::max()};
            fftw_complex*  in_        {nullptr};
            fftw_complex*  out_       {nullptr};
            fftw_plan      plan_      {nullptr};

            /*====================================================================
            Destructor
            --------------------------------------------------------------------*/
            ~FFTPlanClass()
            {
                if(plan_ != nullptr)  {fftw_destroy_plan(    plan_);}
                if(in_   != nullptr)  {fftw_free(            in_);}
                if(out_  != nullptr)  {fftw_free(            out_);}
            }
            /*====================================================================
            Default Constructor
            --------------------------------------------------------------------*/
            FFTPlanClass() =  default;
            /*====================================================================
            Constructor
            --------------------------------------------------------------------*/
            FFTPlanClass(const std::size_t   nfft)
            {

                // allocating nfft
                this->nfft_  = nfft;

                // allocating input-region
                in_     =   reinterpret_cast<fftw_complex*>(
                    fftw_malloc(nfft_ *   sizeof(fftw_complex))
                );
                out_    =   reinterpret_cast<fftw_complex*>(
                    fftw_malloc(nfft_ *   sizeof(fftw_complex))
                );
                if(!in_ || !out_) {throw std::runtime_error("FILE: FFTPlanClass.hpp |
                    CLASS: FFTPlanClass | REPORT: in-out allocation failed");}

                // creating plan
                plan_   =   fftw_plan_dft_1d(
```

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

```
114             in_     =   reinterpret_cast<fftw_complex*>(
115                 fftw_malloc(nfft_ *   sizeof(fftw_complex))
116             );
117             out_    =   reinterpret_cast<fftw_complex*>(
118                 fftw_malloc(nfft_ *   sizeof(fftw_complex))
119             );
120             if(!in_ || !out_)   {throw std::runtime_error("FILE: FFTPlanClass.hpp |
                    CLASS: FFTPlanClass | FUNCTION: Copy-Assignment | REPORT: in-out
                    allocation failed");}
121
122             // copying contents
123             cout << format("\t\t FFTPlanClass& operator=(const FFTPlanClass& other) |
                    nfft_ = {}\n", nfft_);
124             std::memcpy(in_, other.in_, nfft_ * sizeof(fftw_complex));
125             std::memcpy(out_, other.out_, nfft_ * sizeof(fftw_complex));
126
127             // creating engine
128             plan_   =   fftw_plan_dft_1d(
129                 static_cast<int>(nfft_),
130                 in_,
131                 out_,
132                 FFTW_FORWARD,
133                 FFTW_MEASURE
134             );
135             if(!plan_)  {throw std::runtime_error("FILE: FFTPlanClass.hpp | CLASS:
                    FFTPlanClass | FUNCTION: Copy-Assignment | REPORT: plan-creation
                    failed");}
136
137             // returning
138             return *this;
139         }
140         /*=====================================================================
141         Move Constructor
142         ---------------------------------------------------------------------*/
143         FFTPlanClass(FFTPlanClass&& other)
144             :   nfft_(          other.nfft_),
145                 in_(            other.in_),
146                 out_(           other.out_),
147                 plan_(          other.plan_)
148         {
149             // resetting the others
150             other.nfft_       =   0;
151             other.in_         =   nullptr;
152             other.out_        =   nullptr;
153             other.plan_       =   nullptr;
154         }
155         /*=====================================================================
156         Move Assignment
157         ---------------------------------------------------------------------*/
158         FFTPlanClass&  operator=(FFTPlanClass&& other)
159         {
160             // self-assignment check
161             if  (this   ==  &other)    {return *this;}
162
163             // cleaning up existing resources
164             if(plan_ != nullptr)  {fftw_destroy_plan(   plan_);}
165             if(in_   != nullptr)  {fftw_free(           in_);}
166             if(out_  != nullptr)  {fftw_free(           out_);}
167
```

```
168             // Copying-values and changing pointers
169             nfft_             =   other.nfft_;
170             cout << format("\t\t FFTPlanClass's MOVE assignment | nfft_ = {}\n",
                    nfft_);
171             in_               =   other.in_;
172             out_              =   other.out_;
173             plan_             =   other.plan_;
174
175             // resetting source-members
176             other.nfft_       =   0;
177             other.in_         =   nullptr;
178             other.out_        =   nullptr;
179             other.plan_       =   nullptr;
180
181             // returning
182             return *this;
183         }
184         /*=====================================================================
185         Running fft
186         ---------------------------------------------------------------------*/
187         std::vector<destinationType>
188         fft(const  std::vector<sourceType>& input_vector)
189         {
190             // throwing an error
191             if  (input_vector.size()  >   nfft_){
192                 cout << format("input_vector.size() = {}, nfft_ = {}\n",
193                               input_vector.size(),
194                                nfft_);
195                 throw std::runtime_error("FILE: FFTPlanClass.hpp | CLASS: FFTPlanClass
                        | FUNCTION: fft() | REPORT: input-vector size is greater than
                        NFFT");
196             }
197
198
199             // copying inputs
200             for(std::size_t index = 0; index < input_vector.size(); ++index)
201             {
202                 if  constexpr(
203                     std::is_floating_point_v<sourceType>
204                 ){
205                     in_[index][0]  =   input_vector[index];
206                     in_[index][1]  =   0;
207                 }
208                 else if constexpr(
209                     std::is_same_v<sourceType, std::complex<float>> ||
210                     std::is_same_v<sourceType, std::complex<double>>
211                 ){
212                     in_[index][0]  =   input_vector[index].real();
213                     in_[index][1]  =   input_vector[index].imag();
214                 }
215             }
216
217             // executing fft
218             fftw_execute(plan_);
219
220             // copying results to output-vector
221             std::vector<destinationType> output_vector(nfft_);
222             for(std::size_t index = 0; index < nfft_; ++index){
223                 if  constexpr(
```

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

```
            {
                if constexpr(
                    std::is_same_v<   destinationType, std::complex<double> >   ||
                    std::is_same_v<   destinationType, std::complex<float> >
                ){
                    output_vector[index]  =   destinationType(
                        out_[index][0] * (1.00 / std::sqrt(nfft_)),
                        out_[index][1] * (1.00 / std::sqrt(nfft_))
                    );
                }
                else if constexpr(
                    std::is_floating_point_v<destinationType>
                ){
                    output_vector[index]  =   destinationType(
                        std::sqrt(
                            std::pow(out_[index][0] * (1.00 / std::sqrt(nfft_)), 2) + \
                            std::pow(out_[index][1] * (1.00 / std::sqrt(nfft_)), 2)
                        )
                    );
                }
            }

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

A.4 IFFTPlanClass

```
#pragma once
namespace  svr    {

    template   <typename T>
    concept IFFTPlanClassSourceDestinationType = \
        std::is_floating_point_v<T> ||
        (
            std::is_class_v<T> &&
            std::is_floating_point_v<typename T::value_type>
        );
    template   <
        IFFTPlanClassSourceDestinationType sourceType,
        IFFTPlanClassSourceDestinationType destinationType
    >
    class  IFFTPlanClass
    {
        public:
            std::size_t       nfft_;
            fftw_complex*     in_;
            fftw_complex*     out_;
            fftw_plan         plan_;

            /*================================================================
            Destructor
            ----------------------------------------------------------------*/
            ~IFFTPlanClass()
```

```
27          {
28              if(plan_ != nullptr)  {fftw_destroy_plan(    plan_);}
29              if(in_   != nullptr)  {fftw_free(            in_);}
30              if(out_  != nullptr)  {fftw_free(            out_);}
31          }
32          /*=====================================================================
33          Constructor
34          ---------------------------------------------------------------------*/
35          IFFTPlanClass(const std::size_t nfft): nfft_(nfft)
36          {
37              // allocating space
38              in_     =   reinterpret_cast<fftw_complex*>(fftw_malloc(nfft_ *
                    sizeof(fftw_complex)));
39              out_    =   reinterpret_cast<fftw_complex*>(fftw_malloc(nfft_ *
                    sizeof(fftw_complex)));
40              if(!in_ || !out_)   {throw std::runtime_error("in_, out_ creation
                    failed");}
41
42              // creating plan
43              plan_   =   fftw_plan_dft_1d(
44                  static_cast<int>(nfft_),
45                  in_,
46                  out_,
47                  FFTW_BACKWARD,
48                  FFTW_MEASURE
49              );
50              if(!plan_)          {throw std::runtime_error("File: FFTPlanClass.hpp |
                    Class: IFFTPlanClass | report: plan-creation failed");}
51          }
52          /*=====================================================================
53          Copy Constructor
54          ---------------------------------------------------------------------*/
55          IFFTPlanClass(const   IFFTPlanClass&    other)
56          {
57              // allocating space
58              nfft_   =   other.nfft_;
59              in_     =   reinterpret_cast<fftw_complex*>(fftw_malloc(nfft_ *
                    sizeof(fftw_complex)));
60              out_    =   reinterpret_cast<fftw_complex*>(fftw_malloc(nfft_ *
                    sizeof(fftw_complex)));
61              if (!in_ || !out_) {throw std::runtime_error("FILE: FFTPlanClass.hpp |
                    Class: IFFTPlanClass | Function: Copy-Constructor | Report: in-out
                    plan creation failed");}
62
63              // copying contents
64              std::memcpy(in_,  other.in_, nfft_ * sizeof(fftw_complex));
65              std::memcpy(out_, other.out_, nfft_ * sizeof(fftw_complex));
66
67              // creating a new plan since its more of an engine
68              plan_   =   fftw_plan_dft_1d(
69                  static_cast<int>(nfft_),
70                  in_,
71                  out_,
72                  FFTW_BACKWARD,
73                  FFTW_MEASURE
74              );
75              if(!plan_)  {throw std::runtime_error("FILE: FFTPlanClass.hpp | Class:
                    IFFTPlanClass | Function: Copy-Constructor | Report: plan-creation
                    failed");}
```

```
76
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_;
93              in_     =   reinterpret_cast<fftw_complex*>(fftw_malloc(nfft_ *
                    sizeof(fftw_complex)));
94              out_    =   reinterpret_cast<fftw_complex*>(fftw_malloc(nfft_ *
                    sizeof(fftw_complex)));
95              if (!in_ || !out_) {throw std::runtime_error("FILE: FFTPlanClass.hpp |
                    Class: IFFTPlanClass | Function: Copy-Constructor | Report: in-out
                    plan creation failed");}
96
97              // copying contents
98              std::memcpy(in_,  other.in_, nfft_ * sizeof(fftw_complex));
99              std::memcpy(out_, other.out_, nfft_ * sizeof(fftw_complex));
100
101             // creating a new plan since its more of an engine
102             plan_   =   fftw_plan_dft_1d(
103                 static_cast<int>(nfft_),
104                 in_,
105                 out_,
106                 FFTW_BACKWARD,
107                 FFTW_MEASURE
108             );
109             if(!plan_)  {throw std::runtime_error("FILE: FFTPlanClass.hpp | Class:
                    IFFTPlanClass | Function: Copy-Constructor | Report: plan-creation
                    failed");}
110
111             // returning
112             return *this;
113
114         }
115         /*=====================================================================
116         Move Constructor
117         ---------------------------------------------------------------------*/
118         IFFTPlanClass(IFFTPlanClass&& other) noexcept
119             :   nfft_( other.nfft_),
120                 in_(   other.in_),
121                 out_(  other.out_),
122                 plan_( other.plan_)
123         {
124             // resetting the source object
125             other.nfft_    =   0;
126             other.in_      =   nullptr;
127             other.out_     =   nullptr;
128             other.plan_    =   nullptr;
```

```
129             }
130             /*=====================================================================
131             Move-Assignment
132             ---------------------------------------------------------------------*/
133             IFFTPlanClass& operator=(IFFTPlanClass&& other) noexcept
134             {
135                 // self-assignment check
136                 if(this == &other)    {return *this;}
137
138                 // cleaning up existing
139                 if(plan_ != nullptr)  {fftw_destroy_plan(    plan_);}
140                 if(in_   != nullptr)  {fftw_free(            in_);}
141                 if(out_  != nullptr)  {fftw_free(            out_);}
142
143                 // Copying values and changing pointers
144                 nfft_      =   other.nfft_;
145                 in_        =   other.in_;
146                 out_       =   other.out_;
147                 plan_      =   other.plan_;
148
149                 // resetting the source-object
150                 other.nfft_    =   0;
151                 other.in_      =   nullptr;
152                 other.out_     =   nullptr;
153                 other.plan_    =   nullptr;
154
155                 // returning
156                 return *this;
157             }
158             /*=====================================================================
159             Running
160             ---------------------------------------------------------------------*/
161             std::vector<destinationType>
162             ifft(const std::vector<sourceType>& input_vector)
163             {
164                 // throwing an error
165                 if (input_vector.size() > nfft_)
166                     throw std::runtime_error("File: FFTPlanClass | Class: IFFTPlanClass |
                        Function: ifft() | Report: size of vector > nfft ");
167
168                 // copy input into fftw buffer
169                 for(std::size_t index = 0; index < nfft_; ++index)
170                 {
171                     if  constexpr(
172                         std::is_same_v<   sourceType, std::complex<double> >  ||
173                         std::is_same_v<   sourceType, std::complex<float>  >
174                     ){
175                         in_[index][0]  =   input_vector[index].real();
176                         in_[index][1]  =   input_vector[index].imag();
177                     }
178                     else if constexpr(
179                         std::is_floating_point_v< sourceType >
180                     ){
181                         in_[index][0]  =   input_vector[index];
182                         in_[index][1]  =   0;
183                     }
184                 }
185
186                 // execute ifft
```

```
187                 fftw_execute(plan_);
188
189                 // normalize output
190                 std::vector<destinationType> output_vector(nfft_);
191                 for(std::size_t index = 0; index < nfft_; ++index){
192                     if constexpr(
193                         std::is_floating_point_v< destinationType >
194                     ){
195                         output_vector[index]  =
                                static_cast<destinationType>(out_[index][0]/nfft_);
196                     }
197                     else if constexpr(
198                         std::is_same_v<   destinationType, std::complex<double> > ||
199                         std::is_same_v<   destinationType, std::complex<float> >
200                     ){
201                         output_vector[index][0] = destinationType(
202                             out_[index][0]/nfft_,
203                             out_[index][1]/nfft_
204                         );
205                     }
206                 }
207
208                 // returning
209                 return std::move(output_vector);
210             }
211             /*====================================================================
212             Running - proper bases change
213             --------------------------------------------------------------------*/
214             std::vector<destinationType>
215             ifft_l2_conserved(const std::vector<sourceType>& input_vector)
216             {
217                 // throwing an error
218                 if (input_vector.size() > nfft_)
219                     throw std::runtime_error("File: FFTPlanClass | Class: IFFTPlanClass |
                            Function: ifft() | Report: size of vector > nfft ");
220
221                 // copy input into fftw buffer
222                 for(std::size_t index = 0; index < nfft_; ++index)
223                 {
224                     if  constexpr(
225                         std::is_same_v<   sourceType, std::complex<double> >  ||
226                         std::is_same_v<   sourceType, std::complex<float> >
227                     ){
228                         in_[index][0]  =   input_vector[index].real();
229                         in_[index][1]  =   input_vector[index].imag();
230                     }
231                     else if constexpr(
232                         std::is_floating_point_v<sourceType>
233                     ){
234                         in_[index][0]  =   input_vector[index];
235                         in_[index][1]  =   0;
236                     }
237                 }
238
239                 // execute ifft
240                 fftw_execute(plan_);
241
242                 // normalize output
243                 std::vector<destinationType> output_vector(nfft_);
```

```
244                 for(std::size_t index = 0; index < nfft_; ++index)
245                 {
246                     if constexpr(
247                         std::is_floating_point_v<destinationType>
248                     ){
249                         output_vector[index]  =
                                static_cast<destinationType>(out_[index][0] *
                                1.00/std::sqrt(nfft_));
250                     }
251                     else if constexpr(
252                         std::is_same_v<   destinationType, std::complex<double> > ||
253                         std::is_same_v<   destinationType, std::complex<float> >
254                     ){
255                         output_vector[index][0] = destinationType(
256                             out_[index][0] *  1.00/std::sqrt(nfft_),
257                             out_[index][1] *  1.00/std::sqrt(nfft_)
258                         );
259                     }
260                 }
261 
262                 // returning
263                 return std::move(output_vector);
264             }
265     };
266 }
```

A.5 FFT Plan Pool

```
1  #pragma once
2  namespace  svr    {
3      template   <typename  T>
4      concept FFTPlanUniformPoolSourceDestinationType = \
5          std::is_floating_point_v<T>  ||
6          (
7              std::is_class_v<T>       &&
8              std::is_floating_point_v<typename T::value_type>
9          );
10     template   <
11         FFTPlanUniformPoolSourceDestinationType sourceType,
12         FFTPlanUniformPoolSourceDestinationType destinationType
13     >
14     class FFTPlanUniformPool {
15     public:
16         /*=========================================================================
17         Handle to Plan
18         -------------------------------------------------------------------------*/
19         struct AccessPairs
20         {
21             /*=====================================================================
22             Members
23             ---------------------------------------------------------------------*/
24             svr::FFTPlanClass<sourceType, destinationType>& plan;
25             std::unique_lock<std::mutex>                    lock;
26 
27             /*=====================================================================
28             Special Members
```

```
29          ------------------------------------------------------------------*/
30          AccessPairs() = delete;
31          AccessPairs(
32              svr::FFTPlanClass<sourceType, destinationType>& plan_arg,
33              std::mutex&                                     plan_mutex
34          )   :   plan(plan_arg), lock(plan_mutex) {}
35          AccessPairs(
36              svr::FFTPlanClass<sourceType, destinationType>& plan_arg,
37              std::unique_lock<std::mutex>&&                  lock_arg
38          ): plan(plan_arg), lock(std::move(lock_arg)) {}
39          AccessPairs(const AccessPairs&  other)            =  delete;
40          AccessPairs&  operator=(const AccessPairs& other) =  delete;
41          AccessPairs(AccessPairs&& other)                  =  delete;
42          AccessPairs&  operator=(AccessPairs&& other)      =  delete;
43      };
44
45      /*====================================================================
46      Core Members
47      --------------------------------------------------------------------*/
48      std::vector<svr::FFTPlanClass<sourceType, destinationType>> plans;
49      std::vector<std::mutex>                                     mutexes;
50
51      /*====================================================================
52      Special-Members
53      --------------------------------------------------------------------*/
54      FFTPlanUniformPool()                                            =  default;
55      FFTPlanUniformPool(const std::size_t   num_plans,
56                         const   std::size_t   nfft)
57      {
58          // reserving space
59          plans.reserve(num_plans);
60          for(auto i = 0; i < num_plans; ++i){
61              plans.emplace_back(nfft);
62          }
63
64          // creating a vector of mutexes
65          mutexes = std::move(std::vector<std::mutex>(num_plans));
66      }
67      FFTPlanUniformPool(const FFTPlanUniformPool& other)       =  delete;
68      FFTPlanUniformPool& operator=(const FFTPlanUniformPool& other) = delete;
69      FFTPlanUniformPool(FFTPlanUniformPool&& other)            =  default;
70      FFTPlanUniformPool& operator=(FFTPlanUniformPool&& other) =  default;
71
72      /*====================================================================
73      Function to fetch a plan
74          > searches for a free-plan
75          > if found, locks the plan
76              > return the handle to the plan
77      --------------------------------------------------------------------*/
78      AccessPairs fetch_plan() {
79          const int num_rounds = 12;
80          for (int round = 0; round < num_rounds; ++round) {
81              for (int i = 0; i < mutexes.size(); ++i) {
82
83                  std::unique_lock<std::mutex> curr_lock(
84                      mutexes[i],
85                      std::try_to_lock
86                  );
87                  if (curr_lock.owns_lock())
```

```
                    return AccessPairs(plans[i], std::move(curr_lock));

                }
            }
            throw std::runtime_error(
                "FILE: FFTPlanPoolClass.hpp | CLASS: FFTPlanUniformPool | FUNCTION:
                    fetch_plan() | "
                "Report: No plans available despite num_rounds rounds of searching");
        }
    };
}
```

A.6 IFFT Plan Pool

```
#pragma once

/*============================================================================
Dependencies
----------------------------------------------------------------------------*/

namespace  svr    {

    template   <typename  T>
    concept    IFFTPlanUniformPoolSourceDestinationType = \
        std::is_floating_point_v<T>  ||
        (
            std::is_class_v<T> &&
            std::is_floating_point_v<typename T::value_type>
        );
    template   <
        IFFTPlanUniformPoolSourceDestinationType sourceType,
        IFFTPlanUniformPoolSourceDestinationType destinationType
    >
    class IFFTPlanUniformPool
    {
        public:
            /*====================================================================
            Structure used for interfacing to plans
            --------------------------------------------------------------------*/
            struct AccessPairs
            {
                /*================================================================
                Core Members
                ----------------------------------------------------------------*/
                svr::IFFTPlanClass<sourceType, destinationType>&  plan;
                std::unique_lock<std::mutex>                      lock;

                /*================================================================
                Special Members
                ----------------------------------------------------------------*/
                AccessPairs()                                     =   delete;
                AccessPairs(
                    svr::IFFTPlanClass<sourceType, destinationType>& plan_arg,
                    std::mutex&                                   plan_mutex_arg
                ):  plan(plan_arg), lock(plan_mutex_arg) {}
                AccessPairs(
```

```
43                  svr::IFFTPlanClass<sourceType, destinationType>& plan_arg,
44                  std::unique_lock<std::mutex>&&                   lock_arg
45              ):  plan(plan_arg), lock(std::move(lock_arg)) {}
46              AccessPairs(const AccessPairs&  other)            = delete;
47              AccessPairs&  operator=(const AccessPairs& other) = delete;
48              AccessPairs(AccessPairs&& other)                  = delete;
49              AccessPairs&  operator=(AccessPairs&& other)      = delete;
50          };
51
52          /*====================================================================
53          Core Members
54          --------------------------------------------------------------------*/
55          std::vector<  svr::IFFTPlanClass<sourceType, destinationType> > plans;
56          std::vector<  std::mutex                                    >  mutexes;
57
58          /*====================================================================
59          Special Members
60          --------------------------------------------------------------------*/
61          IFFTPlanUniformPool()                       =  default;
62          IFFTPlanUniformPool(const std::size_t num_plans,
63                              const  std::size_t nfft)
64          {
65              // reserving space
66              plans.reserve(num_plans);
67              for(auto i = 0; i < num_plans; ++i)
68                  plans.emplace_back(nfft);
69
70              // creating vector of mutexes
71              mutexes    =  std::vector<std::mutex>(num_plans);
72          }
73          IFFTPlanUniformPool(const IFFTPlanUniformPool& other)            =  delete;
74          IFFTPlanUniformPool&  operator=(const IFFTPlanUniformPool& other) =  delete;
75          IFFTPlanUniformPool(IFFTPlanUniformPool&& other)                 =  default;
76          IFFTPlanUniformPool&  operator=(IFFTPlanUniformPool&& other)     =  default;
77
78          /*====================================================================
79          Member-Functions
80          --------------------------------------------------------------------*/
81          AccessPairs fetch_plan()
82          {
83              // setting the number of rounds to take
84              const  int num_rounds {12};
85
86              // performing rounds
87              for(auto round = 0; round < num_rounds; ++round)
88              {
89                  // going through vector mutexes
90                  for(auto i =0; i < mutexes.size(); ++i)
91                  {
92                      // trying to lock current mutex
93                      std::unique_lock<std::mutex> curr_lock(mutexes[i],
                            std::try_to_lock);
94
95                      // if our lock contains the mutex, returning the plan and lock
96                      if (curr_lock.owns_lock())
97                          return AccessPairs(plans[i], std::move(curr_lock));
98                  }
99              }
100
```

```
                // throwing error
                throw std::runtime_error("FILE: IFFTPlanPoolClass.hpp | CLASS:
                    IFFTPlanUniformPool | REPORT: COULDN'T FIND ANY AVAILABLE PLANS");
            }
    };
}
```

A.7 FFT Plan Pool Handle

```
#pragma once

/*==========================================================================
Dependencies
--------------------------------------------------------------------------*/
#include "FFTPlanPoolClass.hpp"

namespace  svr
{
    template   <typename  T>
    concept    FFTPlanUniformPoolHandleSourceDestinationType = \
        std::is_floating_point_v<T>  ||
        (
            std::is_class_v<T>       &&
            std::is_floating_point_v<typename T::value_type>
        );
    template   <
        FFTPlanUniformPoolHandleSourceDestinationType sourceType,
        FFTPlanUniformPoolHandleSourceDestinationType destinationType
    >
    struct FFTPlanUniformPoolHandle
    {
        /*======================================================================
        Core Members
        ----------------------------------------------------------------------*/
        svr::FFTPlanUniformPool<sourceType, destinationType> uniform_pool;
        std::mutex                                           mutex;
        std::size_t                                          num_plans;
        std::size_t                                          nfft;

        /*======================================================================
        Special Member-functions
        ----------------------------------------------------------------------*/
        FFTPlanUniformPoolHandle()   =   default;
        FFTPlanUniformPoolHandle(const std::size_t num_plans_arg,
                                 const  std::size_t nfft_arg)
            :   uniform_pool(num_plans_arg, nfft_arg),
                num_plans(num_plans_arg),
                nfft(nfft_arg)  {}
        FFTPlanUniformPoolHandle(const FFTPlanUniformPoolHandle& other)      = delete;
        FFTPlanUniformPoolHandle& operator=(const FFTPlanUniformPoolHandle& other) =
            delete;
        FFTPlanUniformPoolHandle(FFTPlanUniformPoolHandle&& other)           = default;
        FFTPlanUniformPoolHandle& operator=(FFTPlanUniformPoolHandle&& other) = default;

        /*======================================================================
        Member Functions
```

```
47          ------------------------------------------------------------------------*/
48          auto    lock()
49          {
50              return std::unique_lock<std::mutex>(this->mutex);
51          }
52      };
53  }
```

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   <typename  T>
12      concept    IFFTPlanUniformPoolHandleSourceDestinationType = \
13          std::is_floating_point_v<T>  ||
14          (
15              std::is_class_v<T> &&
16              std::is_floating_point_v<typename T::value_type>
17          );
18      template   <
19          IFFTPlanUniformPoolHandleSourceDestinationType sourceType,
20          IFFTPlanUniformPoolHandleSourceDestinationType destinationType
21      >
22      struct IFFTPlanUniformPoolHandle
23      {
24          /*========================================================================
25          Members
26          ------------------------------------------------------------------------*/
27          IFFTPlanUniformPool<  sourceType,
28                                destinationType   >       uniform_pool;
29          std::mutex                                      mutex;
30          std::size_t                                     num_plans;
31          std::size_t                                     nfft;
32
33          /*========================================================================
34          Special Member Functions
35          ------------------------------------------------------------------------*/
36          IFFTPlanUniformPoolHandle()  =   default;
37          IFFTPlanUniformPoolHandle(const std::size_t num_plans_arg,
38                                    const std::size_t nfft_arg)
39              :   uniform_pool(         num_plans_arg, nfft_arg),
40                  num_plans(            num_plans_arg),
41                  nfft(                 nfft_arg)  {}
42          IFFTPlanUniformPoolHandle(const IFFTPlanUniformPoolHandle& other)       = delete;
43          IFFTPlanUniformPoolHandle& operator=(const IFFTPlanUniformPoolHandle& other) =
                delete;
44          IFFTPlanUniformPoolHandle(IFFTPlanUniformPoolHandle&& other)            = delete;
45          IFFTPlanUniformPoolHandle& operator=(IFFTPlanUniformPoolHandle&& other) = delete;
```

```
46
47          /*============================================================================
48          Member Functions
49          ----------------------------------------------------------------------------*/
50          auto    lock()
51          {
52              return std::unique_lock<std::mutex>(this->mutex);
53          }
54
55      };
56  }
```

Appendix B

General Purpose Templated Functions

B.1 abs

```
#pragma once
/*============================================================================
Dependencies
----------------------------------------------------------------------------*/
#include <vector>     // for vectors
#include <algorithm>  // for std::transform

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

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

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

    // applying element-wise abs
    std::transform(input_matrix.begin(),
                   input_matrix.end(),
                   input_matrix.begin(),
```

```
                   [](auto& argx){return std::abs(argx);});

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

B.2 Boolean Comparators

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

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

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

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

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

    // transforming
```

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

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

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

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

B.3 Concatenate Functions

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

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

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

}
/*==============================================================================
```

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

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

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

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

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

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

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

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

}
/*==============================================================================
input = [matrix, vector],
output = [matrix]
-------------------------------------------------------------------------------*/
template <std::size_t axis, typename T>
```

```
auto concatenate(const std::vector<std::vector<T>>& input_matrix,
                 const std::vector<T>               input_vector) -> std::enable_if_t<axis
                     == 0, std::vector<std::vector<T>> >
{
    // creating canvas
    auto    canvas     {input_matrix};

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

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

B.4 Conjugate

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

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

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

B.5 Convolution

```
#pragma once
namespace  svr {

    /*============================================================================
    1D convolution of two vectors
    > implemented through fft
    ----------------------------------------------------------------------------*/
    template   <typename T1, typename T2>
    auto    conv1D(const   std::vector<T1>&   input_vector_A,
                   const   std::vector<T2>&   input_vector_B)
    {
        // resulting type
        using  T3  = decltype(std::declval<T1>() *  std::declval<T2>());

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

```
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 -
                    1);
121             length_left_to_fill  =   ending_index - starting_index;
122
123             // copying to the common-block
124             std::copy(large_vector.begin() + starting_index,
125                       large_vector.begin() + ending_index + 1,
126                       canvas.begin());
127
128             // performing convolution
129             block_conv_output =   svr::conv1D_fftw(canvas,
130                                                    small_vector);
131
132             // discarding edges and writing values
133             std::copy(block_conv_output.begin() + filter_size-2,
```

```
                    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);
        }

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

    }
    /*============================================================================
    Long-signal Conv1D with FFT Plan
    improvements:
        > make an inplace version of this
    ----------------------------------------------------------------------------*/
    template   <
        typename   T,
        std::enable_if_t<
            std::is_floating_point_v<T>
        >
    >
    auto   conv1D_long_prototype(
        const  std::vector<T>&                  input_vector_A,
        const  std::vector<T>&                  input_vector_B,
        svr::FFTPlanClass<T, std::complex<T>>&  fft_plan,
        svr::IFFTPlanClass<std::complex<T>, T>& ifft_plan
    )
    {
        // Error checks
        if (fft_plan.nfft_ != ifft_plan.nfft_)
            throw std::runtime_error("fft_plan.nfft_ != ifft_plan.nfft_");

        // fetching references to large-signal and small-signal
        const  auto&  large_signal_original  {
            input_vector_A.size() >= input_vector_B.size() ?
            input_vector_A        :   input_vector_B
        };
        const  auto&  small_signal   {
            input_vector_A.size() <  input_vector_B.size() ?
            input_vector_A        :   input_vector_B
        };

        // copying
        auto   large_signal  {std::vector<double>(
            input_vector_A.size() + input_vector_B.size() -1
        )};
        std::copy(large_signal_original.begin(),
                  large_signal_original.end(),
                  large_signal.begin());;

        // calculating parameters
        const  auto   signal_size          {large_signal_original.size()};
        const  auto   filter_size          {small_signal.size()};
        const  auto   input_signal_block_size {fft_plan.nfft_ + 1 - filter_size};
        if (input_signal_block_size <= 0)
            throw std::runtime_error("input_signal_block_size <= 0 ");
        const  auto   block_output_length   {fft_plan.nfft_};
        const  auto   num_blocks            {static_cast<int>(
            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)) -
                        (static_cast<int>(filter_size) - 1)
219             };
220             auto    analytical_end    {(i+1)*input_signal_block_size -1};
221             start_index    =   std::max(
222                 static_cast<int>(0),  static_cast<int>(analytical_start)
223             );
224             end_index      =   std::min(
225                 static_cast<int>(signal_size-1),static_cast<int>(analytical_end)
226             ); // [start-index, end-index)
227
228             // copying values
229             signal_block_zero_padded = std::move(std::vector<double>(block_output_length,
                    0.0));
230             std::copy(large_signal.begin() + start_index,
231                       large_signal.begin() +  end_index + 1,
232                       signal_block_zero_padded.begin() + start_index - analytical_start);
233
234             // performing ifft(fft(x) * fft(y))
235             fftw_output    =   ifft_plan.ifft(
236                 fft_plan.fft(signal_block_zero_padded) * filter_FFT
237             );
238
239             // trimming away the first parts (since partial)
240             conv_output = std::vector<double>(fftw_output.begin() + filter_size -
                    1,fftw_output.end());
241
242             // writing to final-output
243             std::copy(conv_output.begin(), conv_output.end(), finaloutput.begin() +
                    output_start_index);
244             output_start_index    += conv_output.size();
```

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

```
302     //     );
303
304     //     // fetching an fft and IFFT plan
305     //     auto    fph_lock       {fft_pool_handle.lock()};
306     //     auto    ifph_lock      {ifft_pool_handle.lock()};
307     //     auto    fft_pair       {fft_pool_handle.uniform_pool.fetch_plan()};
308     //     auto    ifft_pair      {ifft_pool_handle.uniform_pool.fetch_plan()};
309
310     //     // performing ifft(fft(x) * filter-FFT)
311     //     fftw_output    =   ifft_pair.plan.ifft_l2_conserved(
312     //         fft_pair.plan.fft_l2_conserved(signal_block_zero_padded) * filter_FFT
313     //     );
314
315     //     // trimming away the first parts (since partial)
316     //     conv_output = std::vector<T>(
317     //         fftw_output.begin() + filter_size - 1,
318     //         fftw_output.end()
319     //     );
320
321     //     // writing to final-output
322     //     auto    output_start_index = i * (block_output_length - (filter_size - 1));
323     //     std::lock_guard<std::mutex>  output_lock(output_vector_mutex);
324     //     std::copy(
325     //         conv_output.begin(), conv_output.end(),
326     //         output_vector.begin() + output_start_index
327     //     );
328     // }
329
330     template   <
331         FFT_SourceDestination_DataType sourceType,
332         FFT_SourceDestination_DataType destinationType
333     >
334     auto    conv_per_plan(
335         const  int                                                  i,
336         const  int&
                   input_signal_block_size,
337         const  int&                                                 filter_size,
338         const  int&
                   block_output_length,
339         const  std::vector<sourceType>&                             large_signal,
340         std::vector<sourceType>
                   signal_block_zero_padded,
341         svr::FFTPlanUniformPoolHandle< sourceType,     destinationType>& fft_pool_handle,
342         svr::IFFTPlanUniformPoolHandle< destinationType, sourceType>&    ifft_pool_handle,
343         const  std::vector<destinationType>&                        filter_FFT,
344         std::vector<sourceType>                                     fftw_output,
345         std::vector<sourceType>                                     conv_output,
346         std::vector<sourceType>&                                    output_vector,
347         std::mutex&
                   output_vector_mutex,
348         const  auto&                                                signal_size
349     )
350     {
351
352         // calculating bounds
353         auto    analytical_start  {
354             (i*static_cast<int>(input_signal_block_size)) -
                   (static_cast<int>(filter_size) - 1)
355         };
```

```
356         auto    analytical_end    {(i+1)*input_signal_block_size -1};
357         auto start_index    =   std::max(
358             static_cast<int>(0),  static_cast<int>(analytical_start)
359         );
360         auto end_index      =   std::min(
361             static_cast<int>(signal_size-1),static_cast<int>(analytical_end)
362         ); // [start-index, end-index)
363 
364         // copying values
365         signal_block_zero_padded = std::move(std::vector<double>(block_output_length,
                0.0));
366         std::copy(
367             large_signal.begin()             +   start_index,
368             large_signal.begin()             +   end_index      + 1,
369             signal_block_zero_padded.begin() +   start_index    - analytical_start
370         );
371 
372         // fetching an fft and IFFT plan
373         auto    fph_lock       {fft_pool_handle.lock()};
374         auto    ifph_lock      {ifft_pool_handle.lock()};
375         auto    fft_pair       {fft_pool_handle.uniform_pool.fetch_plan()};
376         auto    ifft_pair      {ifft_pool_handle.uniform_pool.fetch_plan()};
377 
378         // performing ifft(fft(x) * filter-FFT)
379         fftw_output    =   ifft_pair.plan.ifft_l2_conserved(
380             fft_pair.plan.fft_l2_conserved(signal_block_zero_padded) * filter_FFT
381         );
382 
383         // trimming away the first parts (since partial)
384         conv_output = std::vector<sourceType>(
385             fftw_output.begin() + filter_size - 1,
386             fftw_output.end()
387         );
388 
389         // writing to final-output
390         auto    output_start_index = i * (block_output_length - (filter_size - 1));
391         std::lock_guard<std::mutex>  output_lock(output_vector_mutex);
392         std::copy(
393             conv_output.begin(), conv_output.end(),
394             output_vector.begin() + output_start_index
395         );
396     }
397 
398 
399     template   <
400         typename T,
401         FFT_SourceDestination_DataType sourceType,
402         FFT_SourceDestination_DataType destinationType,
403         typename    =   std::enable_if_t<
404             std::is_floating_point_v<T>
405         >
406     >
407     auto    conv1D_long_FFTPlanPool(
408         const  std::vector<T>&                                          input_vector_A,
409         const  std::vector<T>&                                          input_vector_B,
410         svr::FFTPlanUniformPoolHandle< sourceType,     destinationType>& fft_pool_handle,
411         svr::IFFTPlanUniformPoolHandle< destinationType, sourceType>&    ifft_pool_handle
412     )
413     {
```

```
414         // Error checks
415         if  (fft_pool_handle.nfft!=ifft_pool_handle.nfft)
416             throw std::runtime_error("FILE: svr_conv.hpp | FUNCTION:
                    conv1D_long_FFTPlanPool | Report: the pool-handles are for different
                    nffts");
417
418         // fetching references to the large signal and small signal
419         const  auto&  large_signal_original {
420             input_vector_A.size() >= input_vector_B.size() ?
421             input_vector_A        :   input_vector_B
422         };
423         const  auto&  small_signal  {
424             input_vector_A.size() <  input_vector_B.size() ?
425             input_vector_A        :   input_vector_B
426         };
427
428         // copying
429         auto   large_signal  {std::vector<double>(
430             input_vector_A.size() + input_vector_B.size() - 1
431         )};
432         std::copy(large_signal_original.begin(),
433                   large_signal_original.end(),
434                   large_signal.begin());
435
436         // calculating some parameters
437         const  auto   signal_size   {large_signal_original.size()};
438         const  auto   filter_size   {small_signal.size()};
439         const  auto   input_signal_block_size {
440             fft_pool_handle.nfft + 1 - filter_size
441         };
442
443         // throwing an error if nfft < filter-size
444         if (fft_pool_handle.nfft < filter_size)
445             throw std::runtime_error("FILE: svr_conv.hpp | FUNCTION:
                    conv1D_long_FFTPlanPool | REPORT: filter is bigger than nfft");
446
447         // throwing an error if number of useful samples is less than zero
448         if (input_signal_block_size <= 0)
449             throw std::runtime_error("FILE: svr_conv.hpp | FUNCTION:
                    conv1D_long_FFTPlanPool | REPORT: input_signal_block_size = 0");
450
451
452         const  auto   block_output_length   {fft_pool_handle.nfft};
453         const  auto   num_blocks            {static_cast<int>(
454             1 + std::ceil((signal_size + filter_size - 2) / input_signal_block_size)
455         )};
456         const  auto   final_output_size     {signal_size + filter_size - 1};
457         const  auto   useful_sample_length  {
458             block_output_length - (filter_size - 1) - (filter_size -1)
459         };
460
461         // parameters for re-use
462         auto   start_index          {static_cast<int>(0)};
463         auto   end_index            {static_cast<int>(0)};
464         auto   output_start_index   {static_cast<int>(0)};
465
466         // calculating fft(filter)
467         auto   filter_zero_padded   {std::vector<double>(block_output_length, 0.0)};
468         std::copy(small_signal.begin(), small_signal.end(), filter_zero_padded.begin());
```

```
469         auto    fph_lock0             {fft_pool_handle.lock()};
470         auto    curr_plan_pair        {fft_pool_handle.uniform_pool.fetch_plan()};
471         auto    pool_num_plans        {fft_pool_handle.num_plans};
472         fph_lock0.unlock();
473         auto    filter_FFT            {
474             curr_plan_pair.plan.fft(
475                 filter_zero_padded
476             )
477         };
478         curr_plan_pair.lock.unlock();
479
480         // allocating space for storing input-blocks
481         auto    signal_block_zero_padded {std::vector<T>(block_output_length, 0.0)};
482         auto    fftw_output              {std::vector<T>()};
483         auto    conv_output              {std::vector<T>()};
484         auto    output_vector            {std::vector<T>(final_output_size,  0.0)};
485         auto    output_vector_mutex      {std::mutex()};
486
487         // creating boost
488         svr::ThreadPool local_pool(pool_num_plans);
489
490         // going through the values
491         for(auto i = 0; i < num_blocks; ++i)
492         {
493             local_pool.push_back(
494                 [
495                     i,
496                     &input_signal_block_size,
497                     &filter_size,
498                     &block_output_length,
499                     &large_signal,
500                     signal_block_zero_padded,
501                     &fft_pool_handle,
502                     &ifft_pool_handle,
503                     &filter_FFT,
504                     fftw_output,
505                     conv_output,
506                     &output_vector,
507                     &output_vector_mutex,
508                     &signal_size
509                 ]{
510                     conv_per_plan<T>(
511                         i,
512                         std::ref(input_signal_block_size),
513                         std::ref(filter_size),
514                         std::ref(block_output_length),
515                         std::ref(large_signal),
516                         signal_block_zero_padded,
517                         fft_pool_handle,
518                         ifft_pool_handle,
519                         filter_FFT,
520                         fftw_output,
521                         conv_output,
522                         std::ref(output_vector),
523                         output_vector_mutex,
524                         signal_size
525                     );
526                 }
527             );
```

```
528         }
529         local_pool.converge();
530 
531         // returning final output
532         return std::move(output_vector);
533         // return output_vector;
534     }
535 
536     /*============================================================================
537     Short-conv1D
538     ----------------------------------------------------------------------------*/
539     template   <typename      T1,
540                 typename      T2>
541     auto    conv1D_short(const  std::vector<T1>& input_vector_A,
542                          const   std::vector<T2>& input_vector_B)
543     {
544         // resulting type
545         using  T3  = decltype(std::declval<T1>() *  std::declval<T2>());
546 
547         // creating canvas
548         auto    canvas_length     {input_vector_A.size() + input_vector_B.size() - 1};
549 
550         // calculating fft of two arrays
551         auto    fft_A      {svr::fft(input_vector_A, canvas_length)};
552         auto    fft_B      {svr::fft(input_vector_B, canvas_length)};
553 
554         // element-wise multiplying the two matrices
555         auto    fft_AB     {fft_A *   fft_B};
556 
557         // finding inverse FFT
558         auto    convolved_result  {ifft(fft_AB)};
559 
560         // returning
561         // return std::move(convolved_result);
562         return convolved_result;
563 
564     }
565 
566 
567     /*============================================================================
568     1D Convolution of a matrix and a vector
569     ----------------------------------------------------------------------------*/
570     template   <typename T>
571     auto    conv1D(const   std::vector<std::vector<T>>& input_matrix,
572                    const   std::vector<T>&              input_vector,
573                    const   std::size_t&                 dim)
574     {
575         // getting dimensions
576         const   auto&   num_rows_matrix       {input_matrix.size()};
577         const   auto&   num_cols_matrix       {input_matrix[0].size()};
578         const   auto&   num_elements_vector   {input_vector.size()};
579 
580         // creating canvas
581         auto    canvas     {std::vector<std::vector<T>>()};
582 
583         // creating output based on dim
584         if (dim == 1)
585         {
586             // performing convolutions row by row
```

```
            for(auto   row = 0; row < num_rows_matrix; ++row)
            {
                cout << format("\t\t row = {}/{}\n", row, num_rows_matrix);
                auto bruh  {conv1D(input_matrix[row], input_vector)};
                auto bruh_real {svr::real(std::move(bruh))};

                canvas.push_back(
                       svr::real(
                           std::move(bruh_real)
                       )
                );
            }
        }
        else{
            std::cerr << "svr_conv.hpp | conv1D | yet to be implemented \n";
        }

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

    }

}
```

B.6 Coordinate Change

```
#pragma once
namespace svr {
    /*============================================================================
    y = cart2sph(vector)
    ----------------------------------------------------------------------------*/
    template <typename T>
    auto    cart2sph(const   std::vector<T>& cartesian_vector){

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

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

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

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

        // moving it back
```

```
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]) *
                180.00/std::numbers::pi};
48          auto    elevation_angles  {svr::atan2(cartesian_vector[2],
49                                                xysplat_lengths) *   180.00/std::numbers::pi};
50          auto    rho_values        {norm(cartesian_vector)};
51
52          // creating tesnor
53          cartesian_vector[0]   =   azimuthal_angles;
54          cartesian_vector[1]   =   elevation_angles;
55          cartesian_vector[2]   =   rho_values;
56      }
57      /*=========================================================================
58      y = cart2sph(input_matrix, dim)
59      -------------------------------------------------------------------------*/
60      template <typename T>
61      auto cart2sph(const   std::vector<std::vector<T>>& input_matrix,
62                    const   std::size_t                  axis)
63      {
64          // fetching dimensions
65          const  auto&  num_rows   {input_matrix.size()};
66          const  auto&  num_cols   {input_matrix[0].size()};
67
68          // checking the axis and dimensions
69          if (axis == 0 && num_rows != 3) {std::cerr << "cart2sph: incorrect num-elements
                \n";}
70          if (axis == 1 && num_cols != 3) {std::cerr << "cart2sph: incorrect num-elements
                \n";}
71
72          // creating canvas
73          auto    canvas     {std::vector<std::vector<T>>(
74              num_rows,
75              std::vector<T>(num_cols, 0)
76          )};
77
78          // if axis = 0, performing operation column-wise
79          if(axis == 0)
80          {
81              for(auto col = 0; col < num_cols; ++col)
82              {
83                  // fetching current column
84                  auto    curr_column   {std::vector<T>({input_matrix[0][col],
85                                                         input_matrix[1][col],
86                                                         input_matrix[2][col]})};
```

```
87
88                  // performing inplace transformation
89                  cart2sph_inplace(curr_column);
90
91                  // storing it back
92                  canvas[0][col]   =   curr_column[0];
93                  canvas[1][col]   =   curr_column[1];
94                  canvas[2][col]   =   curr_column[2];
95              }
96          }
97          // if axis == 1, performing operations row-wise
98          else if(axis == 0)
99          {
100             std::cerr << "cart2sph: yet to be implemented \n";
101         }
102         else
103         {
104             std::cerr << "cart2sph: yet to be implemented \n";
105         }
106
107         // returning
108         return std::move(canvas);
109
110     }
111
112     // ============================================================================
113     template <typename T>
114     auto    sph2cart(const std::vector<T> spherical_vector){
115
116         // creating cartesian vector
117         auto    cartesian_vector  {std::vector<T>(spherical_vector.size(), 0)};
118
119         // populating
120         cartesian_vector[0]   =   spherical_vector[2] * \
121                                   cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
122                                   cos(spherical_vector[0] * std::numbers::pi / 180.00);
123         cartesian_vector[1]   =   spherical_vector[2] * \
124                                   cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
125                                   sin(spherical_vector[0] * std::numbers::pi / 180.00);
126         cartesian_vector[2]   =   spherical_vector[2] * \
127                                   sin(spherical_vector[1] * std::numbers::pi / 180.00);
128
129         // returning
130         return std::move(cartesian_vector);
131     }
132 }
```

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  {
```

```
    // created canvas
    auto    canvas     {input_vector};

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

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

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

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

B.8 Data Structures

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


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

B.9 Editing Index Values

```
#pragma once
```

```cpp
/*=============================================================================
Matlab's equivalent of A[A < 0.5] = 0
-----------------------------------------------------------------------------*/
template <typename T, typename U>
auto edit(std::vector<T>&               input_vector,
          const std::vector<bool>&      bool_vector,
          const    U                    scalar)
{
    // throwing an error
    if (input_vector.size() != bool_vector.size())
        std::cerr << "edit: incompatible size\n";

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

    // no-returns since in-place
}

/*=============================================================================
accumulate version of edit, instead of just placing values

THings to add
    - ensuring template only accepts int, std::size_t and similar for T2
    - bring in histogram method to ensure SIMD
-----------------------------------------------------------------------------*/
template   <typename T1,
            typename T2>
auto    edit_accumulate(std::vector<T1>&        input_vector,
                        const  std::vector<T2>&  indices_to_edit,
                        const  std::vector<T1>&  new_values)
{
    // certain checks
    if (indices_to_edit.size() != new_values.size())
        std::cerr << "svr::edit | edit_accumulate | size-disparity occured \n";

    // going through each and accumulating
    for(auto   i = 0; i < input_vector.size(); ++i){
        const  auto  target_index  {static_cast<std::size_t>(indices_to_edit[i])}; //
        const  auto  new_value     {new_values[i]};
        if (target_index < input_vector.size()){
            input_vector[target_index] = input_vector[target_index] + new_value;
        }
        else{
            // std::cout << "warning: FILE: svr_edit.hpp | FUNCTION: edit_accumulate |
                REPORT: index out of bounds";
        }
    }

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

B.10 Equality

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

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

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

B.11 Exponentiate

```cpp
#pragma once

namespace  svr    {
    /*============================================================================
    y = exp(vector)
    ----------------------------------------------------------------------------*/
    template <typename T>
    auto exp(const std::vector<T>&   input_vector)
    {
        // creating canvas
        auto    canvas     {input_vector};

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

        // returning
        return std::move(canvas);
    }
    /*============================================================================
    y = exp(matrix)
    ----------------------------------------------------------------------------*/
    template   <
        typename   sourceType,
        typename   destinationType,
        typename   =   std::enable_if_t<
            std::is_arithmetic_v<sourceType>
        >
    >
    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-mistmatch\n";}
36          if (nfft <= 0)                  {std::cerr << "size-mistmatch\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>(n
                                                * \
57                                          static_cast<baseType>(signal_index)));
58              }
59              finaloutput[frequency_index] = accumulate_value / nfft_sqrt;
60          }
61
62          // returning
63          return std::move(finaloutput);
64      }
65
66      /*=========================================================================
67      y = fft(std::vector<double> nfft) // specialization
68      -------------------------------------------------------------------------*/
69      #include <fftw3.h>       // for fft
70      template <>
71      auto fft(const std::vector<double>& input_vector,
72               const  std::size_t          nfft)
73      {
74          if (nfft < input_vector.size())
75              throw std::runtime_error("nfft must be >= input_vector.size()");
76          if (nfft <= 0)
77              throw std::runtime_error("nfft must be > 0");
78
79          // FFTW real-to-complex output
80          std::vector<std::complex<double>> output(nfft);
81
82          // Allocate input (double) and output (fftw_complex) arrays
83          double* in        = reinterpret_cast<double*>(
84              fftw_malloc(sizeof(double) * nfft)
85          );
86          fftw_complex* out = reinterpret_cast<fftw_complex*>(
87              fftw_malloc(sizeof(fftw_complex) * (nfft/2 + 1))
88          );
89
90          // Copy input and zero-pad if needed
91          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     template<typename T>
133     auto ifft(const    std::vector<T>&   input_vector)
134     {
135         // fetching data-type
136         using  RType   = fft_result_t<T>;
137         using  baseType   = std::conditional_t<std::is_same_v<T, std::complex<double>>,
138                                                double,
139                                                T>;
140 
141         //  setup
142         auto   nfft       {input_vector.size()};
143 
144         // canvas instantiation
145         std::vector<RType> canvas(nfft);
146         auto   nfft_sqrt      {static_cast<RType>(std::sqrt(nfft))};
147         auto   finaloutput    {std::vector<RType>(nfft, 0)};
148 
149         // 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>(n
                                                     * \
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 |
                        source_index !< input_vector.size()\n";
38              }
39
40          }
41
42          // returning
43          return std::move(canvas);
44      }
45      /*==========================================================================
46      y = index(matrix, mask, dim)
47      ----------------------------------------------------------------------------*/
48      template <
49          typename T1,
50          typename T2,
51          typename   =   std::enable_if_t<
52              (std::is_same_v<T1, double> || std::is_same_v<T1, float>) &&
53              (std::is_same_v<T2, int> ||  std::is_same_v<T2, std::size_t>)
54          >
55      >
56      auto index(const  std::vector<std::vector<T1>>& input_matrix,
57                 const   std::vector<T2>&             indices_to_sample,
58                 const   std::size_t&                 dim)
59      {
60          // fetching dimensions
61          const  auto&  num_rows_matrix   {input_matrix.size()};
62          const  auto&  num_cols_matrix   {input_matrix[0].size()};
63
64          // creating canvas
65          auto    canvas {std::vector<std::vector<T1>>()};
66
67          // if indices are row-indices
68          if (dim == 0){
69
70              // initializing canvas
71              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;
```

```
                });
        }
        else if(dim == 1){
            // initializing canvas
            canvas = std::vector<std::vector<T1>>(
                indices_to_sample.size(),
                std::vector<T1>(num_cols_matrix)
            );

            // filling the canvas
            #pragma omp parallel for
            for(auto row = 0; row < canvas.size(); ++row){
                auto   destination_col {0};
                std::for_each(indices_to_sample.begin(), indices_to_sample.end(),
                            [&row,
                             &input_matrix,
                             &destination_col,
                             &canvas](const auto& source_col){
                                 canvas[row][destination_col++] =
                                     input_matrix[row][source_col];
                            });
            }
        }
        else {
            std::cerr << "svr_index | this dim is not implemented \n";
        }

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

B.15 Linspace

```
/*==========================================================================
Dependencies
--------------------------------------------------------------------------*/
#pragma once
#include <vector>
#include <complex>

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

```
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 +
                static_cast<T>(i)*stepsize;}
58          return std::move(input);
59      };
60      /*=============================================================================
61      -----------------------------------------------------------------------------*/
62      template <typename T, typename U>
63      auto linspace(
64          const    T             startvalue,
65          const    U             endvalue,
66          const    std::size_t   numpoints
67      )
68      {
69          std::vector<double> input(numpoints);
70          auto stepsize = static_cast<double>(endvalue -
                startvalue)/static_cast<double>(numpoints-1);
71          for(int i = 0; i<input.size(); ++i) {input[i] = startvalue + i*stepsize;}
72          return std::move(input);
73      };
74  }
```

B.16 Max

```cpp
#pragma once
/*============================================================================
maximum along dimension 1
----------------------------------------------------------------------------*/
template <std::size_t axis, typename T>
auto   max(const  std::vector<std::vector<T>>&  input_matrix) -> std::enable_if_t<axis
    == 1, std::vector<std::vector<T>> >
{
    // setting up canvas
    auto   canvas
        {std::vector<std::vector<T>>(input_matrix.size(),std::vector<T>(1))};

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

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

B.17 Meshgrid

```cpp
/*============================================================================
Dependencies
----------------------------------------------------------------------------*/
#pragma once
#include <vector> // for std::vector
#include <utility> // for std::pair
#include <complex> // for std::complex


/*============================================================================
mesh-grid when working with 1-values
----------------------------------------------------------------------------*/
template <typename T>
auto meshgrid(const   std::vector<T>&  x,
              const    std::vector<T>&  y)
{

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

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

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

}
/*============================================================================
```

```
meshgrid when working with r-values
-------------------------------------------------------------------------------*/
template <typename T>
auto meshgrid(std::vector<T>&& x,
              std::vector<T>&&  y)
{

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

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

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

}
```

B.18 Minimum

```
#pragma once
/*==============================================================================
minimum along dimension 1
-------------------------------------------------------------------------------*/
template <std::size_t axis, typename T>
auto   min(const  std::vector<std::vector<T>>& input_matrix) -> std::enable_if_t<axis ==
    1, std::vector<std::vector<T>> >
{
    // creating canvas
    auto    canvas
        {std::vector<std::vector<T>>(input_matrix.size(),std::vector<T>(1))};

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

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

B.19 Norm

```
#pragma once
/*==============================================================================
calculating norm for vector
-------------------------------------------------------------------------------*/
template <typename T>
auto norm(const   std::vector<T>&   input_vector)
```

```
{
    return std::sqrt(
        std::inner_product(
            input_vector.begin(), input_vector.end(),
            input_vector.begin(),
            (T)0
        )
    );
}
/*============================================================================
Calculating norm of a complex-vector
----------------------------------------------------------------------------*/
template   <>
auto   norm(const std::vector<std::complex<double>>&  input_vector)
{
    return std::sqrt(
        std::inner_product(
            input_vector.begin(),  input_vector.end(),
            input_vector.begin(),
            static_cast<double>(0),
            std::plus<double>(),
            [](const   auto&  argx,
               const   auto&  argy){
                return static_cast<double>(
                    (argx * std::conj(argy)).real()
                );
            }
        )
    );

}
/*============================================================================
----------------------------------------------------------------------------*/
template <typename T>
auto   norm(const std::vector<std::vector<T>>& input_matrix,
            const  std::size_t                 dim)
{
    // creating canvas
    auto    canvas     {std::vector<std::vector<T>>()};
    const   auto&  num_rows_matrix   {input_matrix.size()};
    const   auto&  num_cols_matrix   {input_matrix[0].size()};

    // along dim 0
    if(dim == 0)
    {
        // allocate canvas
        canvas =   std::vector<std::vector<T>>(
            1,
            std::vector<T>(input_matrix[0].size())
        );

        // performing norm
        auto  accumulate_vector   {std::vector<T>(input_matrix[0].size())};

        // going through each row
        for(auto   row = 0; row < num_rows_matrix; ++row)
        {
            std::transform(input_matrix[row].begin(), input_matrix[row].end(),
                           accumulate_vector.begin(),
```

```
                        accumulate_vector.begin(),
                        [](const auto& argx, auto& argy){
                            return argx*argx + argy;
                        });
        }

        // calculating element-wise square root
        std::for_each(accumulate_vector.begin(), accumulate_vector.end(),
                    [](auto& argx){
                         argx = std::sqrt(argx);
                    });

        // moving to the canvas
        canvas[0]  = std::move(accumulate_vector);
    }
    else if (dim == 1)
    {
        // allocating space in the canvas
        canvas     = std::vector<std::vector<T>>(
            input_matrix[0].size(),
            std::vector<T>(1, 0)
        );

        // going through each column
        for(auto row = 0; row < num_cols_matrix; ++row){
            canvas[row][0] = norm(input_matrix[row]);
        }

    }
    else
    {
        std::cerr << "norm(matrix, dim): dimension operation not defined \n";
    }

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



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

B.20 Division

```
#pragma once
/*================================================================================
element-wise division with scalars
--------------------------------------------------------------------------------*/
template <typename T>
auto operator/(const  std::vector<T>&   input_vector,
              const   T&                input_scalar)
{
```

```
    // creating canvas
    auto    canvas     {input_vector};

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

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

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

    // returning value
    return std::move(canvas);
}
/*============================================================================
element-wise with matrix
----------------------------------------------------------------------------*/
template   <
    typename  T,
    typename   =   std::enable_if_t<
        std::is_floating_point_v<T>
    >
>
auto   operator/(const std::vector<std::vector<T>>& input_matrix,
                 const T                             scalar)
{
    // fetching matrix-dimensions
    const   auto&   num_rows_matrix   {input_matrix.size()};
    const   auto&   num_cols_matrix   {input_matrix[0].size()};

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

    // dividing with values
    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 /
                        static_cast<std::complex<numeratorComplexType>>(input_scalar);
106             });
107     }
108
109     // returning values
110     return std::move(canvas);
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
```

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

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

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

B.21 Addition

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

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

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

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

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

    // If b is bigger, resize 'a' to match
    if (a.size() < b.size())                    {a.resize(b.size());}

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

```
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 columsn 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(),
```

```
                canvas.begin(),
                [&scalar](auto& argx){return argx + scalar;});

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

B.22 Multiplication (Element-wise)

```
#pragma once
/*============================================================================
y = scalar * vector
----------------------------------------------------------------------------*/
template <typename T>
auto    operator*(
    const    T                  scalar,
    const    std::vector<T>&    input_vector
)
{
    // creating canvas
    auto    canvas     {input_vector};
    // performing operation
    std::for_each(canvas.begin(), canvas.end(),
                [&scalar](auto& argx){argx = argx * scalar;});
    // returning
    return std::move(canvas);
}
/*============================================================================
y = scalar * vector
----------------------------------------------------------------------------*/
template <
    typename T1,
    typename T2,
    typename = std::enable_if_t<
        !std::is_same_v<std::decay_t<T1>, std::vector<T2> > &&
        std::is_arithmetic_v<T1>
    >
>
auto operator*(const  T1                scalar,
               const    vector<T2>&       input_vector)
{
    // fetching final-type
    using T3 = decltype(std::declval<T1>() * std::declval<T2>());

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

    // multiplying
    std::transform(
        input_vector.begin(),  input_vector.end(),
        canvas.begin(),
        [&scalar](auto& argx){
            return static_cast<T3>(scalar) * static_cast<T3>(argx);
        });

    // returning
```

```
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-desparity
```

```
106     if (input_vector_A.size() != input_vector_B.size()) {std::cerr << "operator*: size
            disparity \n";}
107 
108     // creating canvas
109     auto    canvas     {input_vector_A};
110 
111     // element-wise multiplying
112     std::transform(input_vector_B.begin(), input_vector_B.end(),
113                    canvas.begin(),
114                    canvas.begin(),
115                    [](const auto& argx, const auto& argy){
116                        return argx * argy;
117                    });
118 
119     // moving it back
120     return std::move(canvas);
121 }
122 /*============================================================================
123 y = vecotr * vector
124 ----------------------------------------------------------------------------*/
125 template <
126     typename T1,
127     typename T2,
128     typename   =   std::enable_if_t<
129         std::is_arithmetic_v<T1> &&
130         std::is_arithmetic_v<T2>
131     >
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
                    in the two input-vectors"
178         );
179 
180     // creating canvas
181     auto    canvas     {std::vector<std::complex<T>>( input_vector_A.size() )};
182 
183     // filling up the canvas
184     std::transform(
185         input_vector_B.begin(), input_vector_B.end(),
186         input_vector_A.begin(),
187         canvas.begin(),
188         [](const   auto&   argx,   const   auto&   argy){
189             return argx + static_cast<std::complex<T>>(argy);
190         }
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,
                 complex-vector)"
250         );
251 
252     // creating canvas
253     auto   canvas     {std::vector<std::complex<T>>(input_vector_A.size())};
254 
255     // filling canvas
256     std::transform(
257         input_vector_A.begin(),  input_vector_A.end(),
258         input_vector_B.begin(),
259         canvas.begin(),
260         [](const   auto&  argx, const auto& argy){
261             return argx * argy;
262         }
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     }
```

```cpp
    return std::move(temp);
}
/*============================================================================
y = matrix * scalar
----------------------------------------------------------------------------*/
template <typename T>
auto    operator*(const   std::vector<std::vector<T>>& input_matrix,
                  const    T                           scalar)
{
    // fetching matrix dimensions
    const  auto&  num_rows_matrix   {input_matrix.size()};
    const  auto&  num_cols_matrix   {input_matrix[0].size()};

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

    // storing the values
    for(auto row = 0; row < num_rows_matrix; ++row)
        std::transform(input_matrix[row].begin(), input_matrix[row].end(),
                       canvas[row].begin(),
                       [&scalar](const auto&  argx){
                           return argx * scalar;
                       });

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

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

```
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 aliasses
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 +=
            static_cast<ResultType>(matA[rowA][i]) +
            static_cast<ResultType>(matB[i][colB]);}
388         return temp;
389     };
390
391     // producing row-column combinations
392     std::vector<std::vector<ResultType>> finaloutput(finalnumrows,
        std::vector<ResultType>(finalnumcols));
393     for(int row = 0; row < finalnumrows; ++row){for(int col = 0; col < finalnumcols;
        ++col){finaloutput[row][col] = rowcolproduct(row, col);}}
```

```
394
395     // returning
396     return finaloutput;
397 }
398 /*=============================================================================
399 y = matrix * vector
400 -----------------------------------------------------------------------------*/
401 template   <
402     typename T,
403     typename   =   std::enable_if_t<
404         std::is_arithmetic_v<T>
405     >
406 >
407 auto operator*(const  std::vector<std::vector<T>>&  input_matrix,
408                const   std::vector<T>&              input_vector)
409 {
410     // fetching dimensions
411     const  auto&  num_rows_matrix   {input_matrix.size()};
412     const  auto&  num_cols_matrix   {input_matrix[0].size()};
413     const  auto&  num_rows_vector   {1};
414     const  auto&  num_cols_vector   {input_vector.size()};
415     const  auto&  max_num_rows      {num_rows_matrix > num_rows_vector ?\
416                                      num_rows_matrix : num_rows_vector};
417     const  auto&  max_num_cols      {num_cols_matrix > num_cols_vector ?\
418                                      num_cols_matrix : num_cols_vector};
419
420     // creating canvas
421     auto    canvas     {std::vector<std::vector<T>>(
422         max_num_rows,
423         std::vector<T>(max_num_cols, 0)
424     )};
425
426     // multiplying column matrix with row matrix
427     if (num_cols_matrix == 1 && num_rows_vector == 1){
428
429         // writing to canvas
430         for(auto   row = 0; row < max_num_rows; ++row)
431             for(auto   col = 0; col < max_num_cols; ++col)
432                 canvas[row][col]   =   input_matrix[row][0]  *   input_vector[col];
433     }
434     /*=========================================================================
435     Multiplying each row with the input-vector
436     -------------------------------------------------------------------------*/
437     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,
                  complex-vector)"
531         );
532 
533     // creating canvas
534     auto    canvas     {std::vector<std::vector<std::complex<T>>>(
535         num_rows_matrix,
536         std::vector<std::complex<T>>(
537             num_cols_matrix
538         )
539     )};
540 
541     // filling the values
542     for(auto row = 0; row < num_rows_matrix; ++row)
543         std::transform(
544             input_matrix[row].begin(), input_matrix[row].end(),
545             input_vector.begin(),
546             canvas[row].begin(),
547             [](const   auto&  argx,  const  auto&  argy){
548                 return static_cast<std::complex<T>>(argx) * argy;
549             }
550         );
551 
552     // returning final-output
553     return std::move(canvas);
554 }
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 }
```

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

B.23 Subtraction

```
#pragma once
/*============================================================================
y = vector - scalar
----------------------------------------------------------------------------*/
template <typename T>
auto operator-(const std::vector<T>& a,
               const T               scalar){
    std::vector<T> temp(a.size());
    std::transform(a.begin(),
                   a.end(),
                   temp.begin(),
                   [scalar](T x){return (x - scalar);});
    return std::move(temp);
}
/*============================================================================
y = vector - vector
----------------------------------------------------------------------------*/
template <typename T>
auto operator-(const  std::vector<T>&   input_vector_A,
               const   std::vector<T>&   input_vector_B)
{
    // throwing error
    if (input_vector_A.size() != input_vector_B.size())
        std::cerr << "operator-(vector, vector): size disparity\n";

    // creating canvas
    const  auto&  num_cols   {input_vector_A.size()};
    auto          canvas     {std::vector<T>()};

    // peforming operations
    std::transform(input_vector_A.begin(), input_vector_A.begin(),
                   input_vector_B.begin(),
                   canvas.begin(),
                    [](const auto& argx, const auto& argy){
                        return argx - argy;
                    });

    // return
    return std::move(canvas);
}
/*============================================================================
y = matrix - matrix
----------------------------------------------------------------------------*/
template <typename T>
auto operator-(const  std::vector<std::vector<T>>&  input_matrix_A,
               const   std::vector<std::vector<T>>&  input_matrix_B)
{
    // fetching dimensions
    const  auto&  num_rows_A {input_matrix_A.size()};
```

```
    const  auto&  num_cols_A {input_matrix_A[0].size()};
    const  auto&  num_rows_B {input_matrix_B.size()};
    const  auto&  num_cols_B {input_matrix_B[0].size()};

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

    // if both matrices are of equal dimensions
    if (num_rows_A == num_rows_B && num_cols_A == num_cols_B)
    {
        // copying one to the canvas
        canvas = input_matrix_A;

        // subtracting
        for(auto row = 0; row < num_rows_B; ++row)
            std::transform(canvas[row].begin(), canvas[row].end(),
                           input_matrix_B[row].begin(),
                           canvas[row].begin(),
                           [](auto& argx, const auto& argy){
                               return argx - argy;
                           });
    }
    // column broadcasting (case 1)
    else if(num_rows_A == num_rows_B && num_cols_B == 1)
    {
        // copying canvas
        canvas =  input_matrix_A;

        // substracting
        for(auto row = 0; row < num_rows_A; ++row){
            std::transform(canvas[row].begin(), canvas[row].end(),
                           canvas[row].begin(),
                           [&input_matrix_B,
                            &row](auto& argx){
                               return argx - input_matrix_B[row][0];
                           });
        }
    }
    else{
        std::cerr << "operator-: not implemented for this case \n";
    }

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

B.24 Printing Containers

```
#pragma once
/*============================================================================
----------------------------------------------------------------------------*/
template<typename T>
void fPrintVector(const vector<T> input){
    for(auto x: input) cout << x << ",";
    cout << endl;
}
```

```
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 -------------------------------------------------------------------------*/
```

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

B.25 Random Number Generation

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

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

    return finaloutput;
}
```

```
/*============================================================================
----------------------------------------------------------------------------*/
template <typename T>
auto rand(const T                argmin,
          const T                argmax,
          const std::vector<int> dimensions)
{

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

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

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

        finaloutput.push_back(temp);
    }

    // returning the finaloutput
    return finaloutput;

}
/*============================================================================
----------------------------------------------------------------------------*/
auto rand(const std::vector<int> dimensions)
{
    using ReturnType = double;

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

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

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

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

}
/*============================================================================
----------------------------------------------------------------------------*/
template <typename T>
```

```
auto rand_complex_double(const T                argmin,
                         const T                argmax,
                         const std::vector<int>& dimensions)
{

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

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

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

    // returning the finaloutput
    return finaloutput;
}
```

B.26 Reshape

```
#pragma once

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

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

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

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

```
    }

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

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

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

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

    // moving it back
    return std::move(canvas);
}
/*============================================================================
Matrix to matrix
----------------------------------------------------------------------------*/
template<typename T>
auto reshape(const std::vector<std::vector<T>>& input_matrix,
             const std::size_t              M,
             const std::size_t              N){

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

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

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

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

```
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,
        std::vector<T>>
6   {
7       // returning the input as is
8       return input_vector;
9   }
10  /*=========================================================================
11  -------------------------------------------------------------------------*/
12  template <std::size_t axis, typename T>
13  auto sum(const std::vector<std::vector<T>>& input_matrix) -> std::enable_if_t<axis == 0,
        std::vector<T>>
14  {
15      // creating canvas
16      auto    canvas     {std::vector<T>(input_matrix[0].size(), 0)};
17
18      // filling up the canvas
19      for(auto row = 0; row < input_matrix.size(); ++row)
20          std::transform(input_matrix[row].begin(), input_matrix[row].end(),
21                         canvas.begin(),
22                         canvas.begin(),
23                         [](auto& argx, auto& argy){return argx + argy;});
24
25      // returning
26      return std::move(canvas);
27
28  }
29  /*=========================================================================
```

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

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

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

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

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

    // creating canvas
    auto   canvas {input_vector_A};

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

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

B.28 Tangent

```
#pragma once
namespace  svr {
    /*============================================================================
    y = tan-inverse(input_vector_A/input_vector_B)
    ----------------------------------------------------------------------------*/
    template <typename T>
    auto atan2(const  std::vector<T>    input_vector_A,
               const   std::vector<T>    input_vector_B)
```

```
    {
        // throw error
        if (input_vector_A.size() != input_vector_B.size())
            std::cerr << "atan2: size disparity\n";

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

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

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

        // moving things back
        return std::move(canvas);
    }
    /*============================================================================
    y = tan-inverse(a/b)
    ----------------------------------------------------------------------------*/
    template <typename T>
    auto atan2(T   scalar_A,
               T   scalar_B)
    {
        return std::atan2(scalar_A, scalar_B);
    }
}
```

B.29 Tiling Operations

```
#pragma once
namespace  svr {
    /*============================================================================
    tiling a vector
    ----------------------------------------------------------------------------*/
    template <typename T>
    auto tile(const   std::vector<T>&       input_vector,
              const   std::vector<size_t>&  mul_dimensions){

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

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

        for(std::size_t row = 0; row < num_rows; ++row){
```

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

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

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

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

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

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

B.30 Transpose

```
#pragma once
/*============================================================================
----------------------------------------------------------------------------*/
template <typename T>
auto transpose(const  std::vector<T>&   input_vector){

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

```

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

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

B.31 Masking

```
#pragma once
namespace  svr {
    /*============================================================================
    y = input_vector[mask == 1]
    ----------------------------------------------------------------------------*/
    template <typename T,
              typename = std::enable_if_t< std::is_arithmetic_v<T>           ||
                                           std::is_same_v<T, std::complex<double>> ||
                                           std::is_same_v<T, std::complex<float>>
                                         >
             >
    auto mask(const   std::vector<T>&   input_vector,
              const    std::vector<bool>& mask_vector)
    {
        // checking dimensionality issues
        if (input_vector.size() != mask_vector.size())
            std::cerr << "mask(vector, mask): incompatible size \n";

        // creating canas
        auto    num_trues  {std::count(mask_vector.begin(),
                                       mask_vector.end(),
                                       true)};
        auto    canvas     {std::vector<T>(num_trues)};

        // copying values
        auto    destination_index {0};
        for(auto i = 0; i <input_vector.size(); ++i)
            if (mask_vector[i] == true)
                canvas[destination_index++] = input_vector[i];

        // returning output
        return std::move(canvas);
    }
    /*============================================================================
    ----------------------------------------------------------------------------*/
    template <typename T>
    auto mask(const   std::vector<std::vector<T>>&     input_matrix,
              const    std::vector<bool>                mask_vector)
    {
        // fetching dimensions
        const  auto&  num_rows_matrix   {input_matrix.size()};
        const  auto&  num_cols_matrix   {input_matrix[0].size()};
        const  auto&  num_cols_vector       {mask_vector.size()};

        // error-checking
```

```
        if (num_cols_matrix != num_cols_vector)
            std::cerr << "mask(matrix, bool-vector): size disparity";

        // building canvas
        auto    num_trues  {std::count(mask_vector.begin(),
                                       mask_vector.end(),
                                       true)};
        auto    canvas     {std::vector<std::vector<T>>(
            num_rows_matrix,
            std::vector<T>(num_cols_vector, 0)
        )};

        // writing values
        #pragma omp parallel for
        for(auto row = 0; row < num_rows_matrix; ++row){
            auto    destination_index {0};
            for(auto col = 0; col < num_cols_vector; ++col)
                if(mask_vector[col] == true)
                    canvas[row][destination_index++] = input_matrix[row][col];
        }

        // returning
        return std::move(canvas);
    }
    /*============================================================================
    Fetch Indices corresponding to mask true's
    ----------------------------------------------------------------------------*/
    auto mask_indices(const std::vector<bool>& mask_vector)
    {
        // creating canvas
        auto    num_trues  {std::count(mask_vector.begin(), mask_vector.end(),
                                       true)};
        auto    canvas     {std::vector<std::size_t>(num_trues)};

        // building canvas
        auto    destination_index {0};
        for(auto i = 0; i < mask_vector.size(); ++i)
            if (mask_vector[i] == true)
                canvas[destination_index++] = i;

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

B.32 Resetting Containers

```
#pragma once
namespace  svr {
    /*============================================================================
    Variadic version of resetting
    ----------------------------------------------------------------------------*/
    template <typename T, typename... Rest>
    void reset(std::vector<T>& first_vector, Rest&... rest_vectors) {
        // Reset the first vector
        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          // peforming 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()};
```

```
        const  auto&  num_cols   {input_matrix[0].size()};

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

        // going through each row
        #pragma omp parallel for
        for(auto row = 0; row < num_rows; ++row)
            std::transform(input_matrix[row].begin(), input_matrix[row].end(),
                           canvas[row].begin(),
                           [](const auto& argx){
                               return argx * argx;
                           });

        // returning
        return std::move(canvas);
    }
    /*============================================================================
    Squaring for scalars
    ----------------------------------------------------------------------------*/
    template <typename T>
    auto square(const T&  scalar)    {return scalar * scalar;}
}
```

B.34 Flooring

```
namespace  svr {
    /*============================================================================
    element-wise flooring of a vector-contents
    ----------------------------------------------------------------------------*/
    template   <typename  T>
    auto   floor(const   std::vector<T>&   input_vector)
    {
        // creating canvas
        auto   canvas     {std::vector<T>(input_vector.size())};

        // filling the canvas
        std::transform(input_vector.begin(), input_vector.end(),
                       canvas.begin(),
                       [](const auto& argx){
                           return std::floor(argx);
                       });

        // returning
        return std::move(canvas);
    }
    /*============================================================================
    element-wise flooring of a vector-contents (in-place)
    ----------------------------------------------------------------------------*/
    template   <typename  T>
    auto   floor_inplace(std::vector<T>&   input_vector)
    {
        // rewriting the contents
```

```
        std::transform(input_vector.begin(), input_vector.end(),
                       input_vector.begin(),
                       [](auto& argx){
                           return std::floor(argx);
                       });
    }
    /*============================================================================
    element-wise flooring of matrix-contents
    ----------------------------------------------------------------------------*/
    template   <typename T>
    auto   floor(const std::vector<std::vector<T>>& input_matrix)
    {
        // fetching dimensions
        const  auto&  num_rows_matrix   {input_matrix.size()};
        const  auto&  num_cols_matrix   {input_matrix[0].size()};

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

        // writing contents
        for(auto row = 0; row < num_rows_matrix; ++row)
            std::transform(input_matrix[row].begin(), input_matrix[row].end(),
                           canvas[row].begin(),
                           [](const auto& argx){
                               return std::floor(argx);
                           });

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

    }
    /*============================================================================
    element-wise flooring of matrix-contents (in-place)
    ----------------------------------------------------------------------------*/
    template   <typename T>
    auto   floor_inplace(std::vector<std::vector<T>>& input_matrix)
    {
        // performing operations
        for(auto row = 0; row < input_matrix.size(); ++row)
            std::transform(input_matrix[row].begin(), input_matrix[row].end(),
                           input_matrix[row].begin(),
                           [](auto& argx){
                               return std::floor(argx);
                           });
    }
}
```

B.35 Squeeze

```
namespace  svr {
    template   <typename  T>
    auto   squeeze(const  std::vector<std::vector<T>>& input_matrix)
    {
```

```
        // fetching dimensions
        const  auto&  num_rows_matrix   {input_matrix.size()};
        const  auto&  num_cols_matrix   {input_matrix[0].size()};

        // check if any dimension is 1
        if (num_rows_matrix == 0 || num_cols_matrix == 0)
            std::cerr << "at least one dimension should be 1";

        auto   final_length   {std::max(num_rows_matrix, num_cols_matrix)};

        // creating canvas
        auto   canvas         {std::vector<T>(final_length)};

        // building canvas
        if (num_rows_matrix == 1)
        {
            // filling canvas
            std::copy(input_matrix[0].begin(), input_matrix[0].end(),
                      canvas.begin());
        }
        else if(num_cols_matrix == 1)
        {
            // filling canvas
            std::transform(input_matrix.begin(), input_matrix.end(),
                           canvas.begin(),
                           [](const auto& argx){
                               return argx[0];
                           });
        }

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

B.36 Tensor Initializations

```
namespace  svr {
    /*============================================================================
    ----------------------------------------------------------------------------*/
    template <typename T>
    auto   zeros(const std::array<std::size_t, 2> input_dimensions)
    {
        // create canvas
        auto   canvas     {std::vector<std::vector<T>>(
            input_dimensions[0],
            std::vector<T>(input_dimensions[1], 0)
        )};

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

B.37 Real part

```cpp
#pragma once
namespace  svr {

    /*================================================================================
    For type-deductions
    --------------------------------------------------------------------------------*/
    template   <typename T>
    struct real_result_type;

    template <> struct real_result_type<std::complex<double>>{
        using type = double;
    };
    template <> struct real_result_type<std::complex<float>>{
        using type = float;
    };
    template <> struct real_result_type<double> {
        using type = double;
    };
    template <> struct real_result_type<float>{
        using type = float;
    };

    template   <typename T>
    using  real_result_t  = typename real_result_type<T>::type;

    /*================================================================================
    Element-wise real() of a vector
    --------------------------------------------------------------------------------*/
    template   <typename T>
    auto   real(const std::vector<T>& input_vector)
    {
        // figure out base-type
        using TCanvas  = real_result_t<T>;

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

        // storing values
        std::transform(input_vector.begin(), input_vector.end(),
                       canvas.begin(),
                       [](const   auto&   argx){
                           return std::real(argx);
                       });

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

B.38 Imaginary part

```cpp
#pragma once
```

```
namespace  svr {

    /*================================================================================
    For type-deductions
    --------------------------------------------------------------------------------*/
    template   <typename T>
    struct imag_result_type;

    template <> struct imag_result_type<std::complex<double>>{
        using type = double;
    };
    template <> struct imag_result_type<std::complex<float>>{
        using type = float;
    };
    template <> struct imag_result_type<double> {
        using type = double;
    };
    template <> struct imag_result_type<float>{
        using type = float;
    };

    template   <typename T>
    using  imag_result_t  = typename imag_result_type<T>::type;

    /*================================================================================
    --------------------------------------------------------------------------------*/
    template   <typename  T>
    auto   imag(const std::vector<T>&   input_vector)
    {
        // figure out base-type
        using TCanvas  = imag_result_t<T>;

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

        // storing values
        std::transform(input_vector.begin(), input_vector.end(),
                       canvas.begin(),
                       [](const   auto&   argx){
                           return std::imag(argx);
                       });

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