Autonomous Underwater Vehicle: A Surveillance Protocol

S.V. Rajendran

October 3, 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

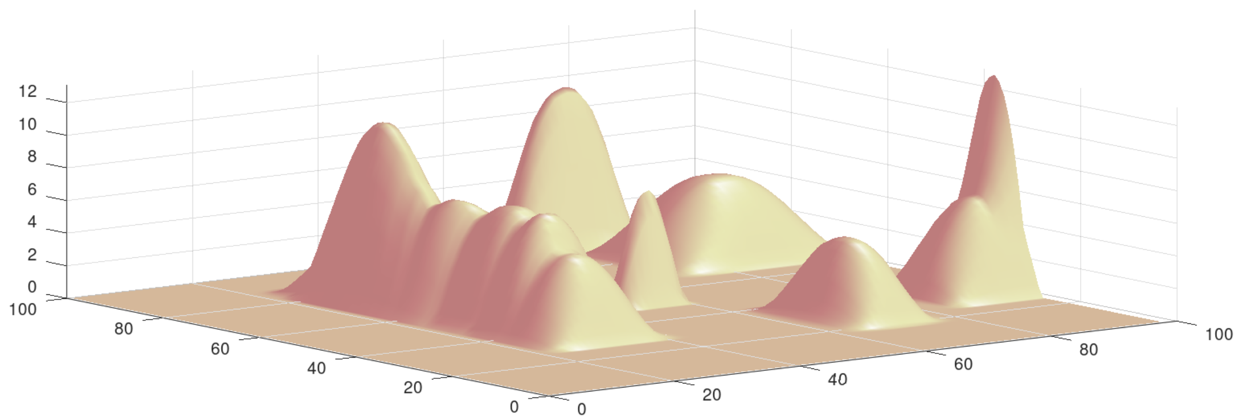
Underwater Environment

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)};

    // setting up coordinates
    auto    xpoints    {linspace<double>(0.00,
                                  bed_width,
                                  bed_width * bed_width_density)};
    auto    ypoints    {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){

        // setting up hill parameters
        auto    num_hills     {10};

        // setting up placement of hills
        auto    points2D             {concatenate<0>(X, Y)};            // verified
        auto    min2D                {min<1, double>(points2D)};        // verified
        auto    max2D                {max<1, double>(points2D)};        // verified
```

```
        auto   hill_2D_center        {min2D + \
                                      rand({2, num_hills}) * (max2D - min2D)}; // verified

        // setup: hill-dimensions
        auto   hill_dimensions_min   {transpose(vector<double>{5, 5, 2})};   // verified
        auto   hill_dimensions_max   {transpose(vector<double>{30, 30, 10})}; // verified
        auto   hill_dimensions       {hill_dimensions_min + \
                                      rand({3, num_hills}) * (hill_dimensions_max -
                                          hill_dimensions_min)};              // verified

        // function-call: hill-creation function
        fCreateHills(hill_2D_center,
                     hill_dimensions,
                     points2D);

        // setting up floor reflectivity
        auto  floorScatter_reflectivity    {std::vector<double>(Y.size(), 1.00)};

        // populating the values of the incoming argument
        scatterers.coordinates    = std::move(points2D);
        scatterers.reflectivity   = std::move(floorScatter_reflectivity);

    }
    else{

        // assigning flat heights
        auto   Z      {std::vector<double>(Y.size(), 0)};

        // setting up floor coordinates
        auto   floorScatter_coordinates    {concatenate<0>(X, Y, Z)};
        auto   floorScatter_reflectivity   {std::vector<double>(Y.size(), 1)};

        // populating the values of the incoming argument
        scatterers.coordinates    = std::move(floorScatter_coordinates);
        scatterers.reflectivity   = std::move(floorScatter_reflectivity);

    }
}
```

Chapter 2

Transmitter

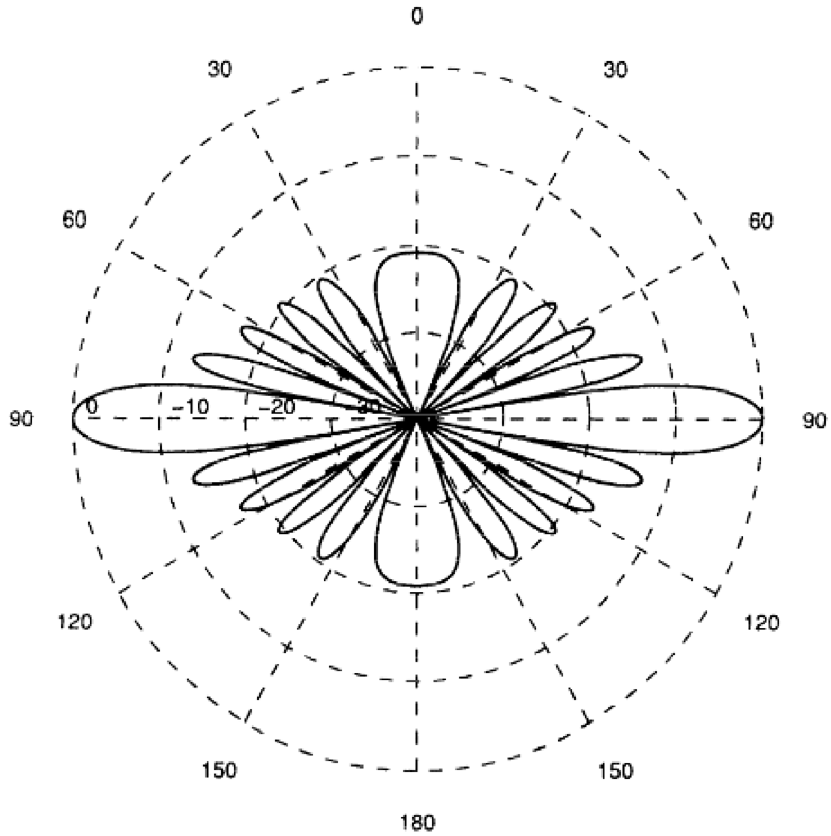


Figure 2.1: Beampattern of a Transmission Uniform Linear Array

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

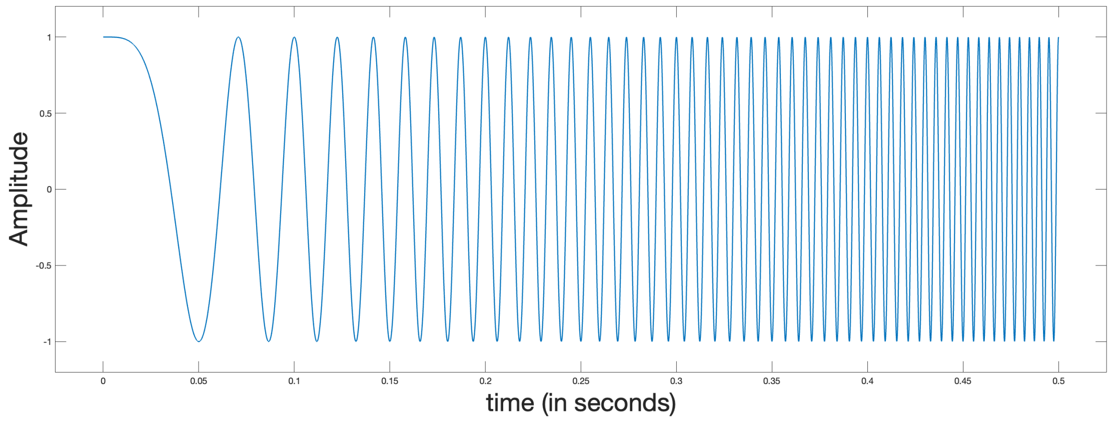


Figure 2.2: Linear Frequency Modulated Wave

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.

```
template <typename T>
class TransmitterClass{
public:

    // A shared pointer to the configuration object
    std::shared_ptr<svr::AUVParameters> config_ptr;

    // physical/intrinsic properties
    std::vector<T>    location;             // location tensor
    std::vector<T>    pointing_direction;   // pointing direction

    // basic parameters
    std::vector<T>    signal;               // transmitted signal (LFM)
    T                 azimuthal_angle;      // transmitter's azimuthal pointing direction
    T                 elevation_angle;      // transmitter's elevation pointing direction
    T                 azimuthal_beamwidth;  // azimuthal beamwidth of transmitter
    T                 elevation_beamwidth;  // elevation beamwidth of transmitter
    T                 range;                // a parameter used for spotlight mode.

    // transmitted signal attributes
    T                 f_low;                // lowest frequency of LFM
    T                 f_high;               // highest frequency of LFM
    T                 fc;                   // center frequency of LFM
    T                 bandwidth;            // bandwidth of LFM
    T                 speed_of_sound {1500};   // speed of sound

    // shadowing properties
    int               azimuthQuantDensity;      // quantization of angles along the azimuth
    int               elevationQuantDensity;    // quantization of angles along the elevation
    T                 rangeQuantSize;           // range-cell size when shadowing
    T                 azimuthShadowThreshold;   // azimuth thresholding
    T                 elevationShadowThreshold; // elevation thresholding
```

```

    // shadowing related
    std::vector<T>    checkbox;           // box indicating whether a scatter for a range-angle pair has been found
    std::vector<std::vector<std::vector<T>>> finalScatterBox; // a 3D tensor where the third dimension represnets the vector length
    std::vector<T> finalReflectivityBox; // to store the reflectivity

    // constructor
    TransmitterClass() = default;

    // Deleting copy constructors/assignment
    TransmitterClass(const TransmitterClass& other)       = delete;
    TransmitterClass& operator=(TransmitterClass& other)  = delete;

    // Creating move-constructor and move-assignment
    TransmitterClass(TransmitterClass&& other)            = default;
    TransmitterClass& operator=(TransmitterClass&& other) = default;

    // member-functions
    auto updatePointingAngle(std::vector<T> AUV_pointing_vector);
    auto subset_scatterers(const ScattererClass<T>& seafloor,
```

2.3 Transmitter Setup Scripts

The following script shows the setup-script

```
template <typename T>
void fTransmitterSetup(TransmitterClass<T>& transmitter_fls,
                       TransmitterClass<T>&  transmitter_portside,
                       TransmitterClass<T>&  transmitter_starboard)
{
    // Setting up transmitter
    T      sampling_frequency    {160e3};              // sampling frequency
    T      f1                    {50e3};               // first frequency of LFM
    T      f2                    {70e3};               // second frequency of LFM
    T      fc                    {(f1 + f2)/2.00};     // finding center-frequency
    T      bandwidth             {std::abs(f2 - f1)};  // bandwidth
    T      pulselength           {5e-2};               // time of recording

    // building LFM
    auto   timearray     {linspace<T>(0.00,
                                      pulselength,
                                      std::floor(pulselength * sampling_frequency))};
    auto   K             {f2 - f1/pulselength}; // calculating frequency-slope
    auto   Signal        {cos(2 * std::numbers::pi * \
                          (f1 + K*timearray) * \
                          timearray)};         // frequency at each time-step, with f1 = 0

    // Setting up transmitter
    auto   location                   {std::vector<T>(3, 0)};      // location of transmitter
    T      azimuthal_angle_fls        {0};                         // initial pointing direction
    T      azimuthal_angle_port       {90};                        // initial
```

```
    pointing direction
    T     azimuthal_angle_starboard      {-90};                        // initial pointing direction

    T     elevation_angle                {-60};                        // initial pointing direction

    T     azimuthal_beamwidth_fls        {20};                         // azimuthal beamwidth of the signal cone
    T     azimuthal_beamwidth_port       {20};                         // azimuthal beamwidth of the signal cone
    T     azimuthal_beamwidth_starboard  {20};                         // azimuthal beamwidth of the signal cone

    T     elevation_beamwidth_fls        {20};                         // elevation beamwidth of the signal cone
    T     elevation_beamwidth_port       {20};                         // elevation beamwidth of the signal cone
    T     elevation_beamwidth_starboard  {20};                         // elevation beamwidth of the signal cone

    int   azimuthQuantDensity            {10};  // number of points, a degree is split into quantization density along azimuth (used for shadowing)
    int   elevationQuantDensity          {10};  // number of points, a degree is split into quantization density along elevation (used for shadowing)
    T     rangeQuantSize                 {10};  // the length of a cell (used for shadowing)

    T     azimuthShadowThreshold         {1};   // azimuth threshold   (in degrees)
    T     elevationShadowThreshold       {1};   // elevation threshold (in degrees)


    // transmitter-fls
    transmitter_fls.location            = location;                    // Assigning location
    transmitter_fls.signal              = Signal;                      // Assigning signal
    transmitter_fls.azimuthal_angle     = azimuthal_angle_fls;         // assigning azimuth angle
    transmitter_fls.elevation_angle     = elevation_angle;             // assigning elevation angle
    transmitter_fls.azimuthal_beamwidth = azimuthal_beamwidth_fls;     // assigning azimuth-beamwidth
    transmitter_fls.elevation_beamwidth = elevation_beamwidth_fls;     // assigning elevation-beamwidth
    // updating quantization densities
    transmitter_fls.azimuthQuantDensity    = azimuthQuantDensity;     // assigning azimuth quant density
    transmitter_fls.elevationQuantDensity  = elevationQuantDensity;   // assigning elevation quant density
    transmitter_fls.rangeQuantSize         = rangeQuantSize;          // assigning range-quantization
    transmitter_fls.azimuthShadowThreshold = azimuthShadowThreshold;  // azimuth-threshold in shadowing
    transmitter_fls.elevationShadowThreshold = elevationShadowThreshold; // elevation-threshold in shadowing
    // signal related
    transmitter_fls.f_low                  = f1;          // assigning lower frequency
    transmitter_fls.f_high                 = f2;          // assigning higher frequency
```

```
    transmitter_fls.fc                     = fc;         // assigning center frequency
    transmitter_fls.bandwidth              = bandwidth;  // assigning bandwidth


    // transmitter-portside
    transmitter_portside.location              = location;                 // Assigning location
    transmitter_portside.signal                = Signal;                   // Assigning signal
    transmitter_portside.azimuthal_angle       = azimuthal_angle_port;     // assigning azimuth angle
    transmitter_portside.elevation_angle       = elevation_angle;          // assigning elevation angle
    transmitter_portside.azimuthal_beamwidth   = azimuthal_beamwidth_port; // assigning azimuth-beamwidth
    transmitter_portside.elevation_beamwidth   = elevation_beamwidth_port; // assigning elevation-beamwidth
    // updating quantization densities
    transmitter_portside.azimuthQuantDensity   = azimuthQuantDensity;      // assigning azimuth quant density
    transmitter_portside.elevationQuantDensity = elevationQuantDensity;    // assigning elevation quant density
    transmitter_portside.rangeQuantSize        = rangeQuantSize;           // assigning range-quantization
    transmitter_portside.azimuthShadowThreshold = azimuthShadowThreshold;  // azimuth-threshold in shadowing
    transmitter_portside.elevationShadowThreshold = elevationShadowThreshold; // elevation-threshold in shadowing
    // signal related
    transmitter_portside.f_low                 = f1;                       // assigning lower frequency
    transmitter_portside.f_high                = f2;                       // assigning higher frequency
    transmitter_portside.fc                    = fc;                       // assigning center frequency
    transmitter_portside.bandwidth             = bandwidth;                // assigning bandwidth


    // transmitter-starboard
    transmitter_starboard.location                = location;                  // assigning location
    transmitter_starboard.signal                  = Signal;                    // assigning signal
    transmitter_starboard.azimuthal_angle         = azimuthal_angle_starboard; // assigning azimuthal signal
    transmitter_starboard.elevation_angle         = elevation_angle;
    transmitter_starboard.azimuthal_beamwidth     = azimuthal_beamwidth_starboard;
    transmitter_starboard.elevation_beamwidth     = elevation_beamwidth_starboard;
    // updating quantization densities
    transmitter_starboard.azimuthQuantDensity     = azimuthQuantDensity;        // assigning azimuth-quant-density
    transmitter_starboard.elevationQuantDensity   = elevationQuantDensity;
    transmitter_starboard.rangeQuantSize          = rangeQuantSize;
    transmitter_starboard.azimuthShadowThreshold  = azimuthShadowThreshold;
    transmitter_starboard.elevationShadowThreshold = elevationShadowThreshold;
    // signal related
    transmitter_starboard.f_low                   = f1;                         // assigning lower frequency
```

```
    transmitter_starboard.f_high                    = f2;                           // assigning higher frequency
    transmitter_starboard.fc                        = fc;                           // assigning center frequency
    transmitter_starboard.bandwidth                 = bandwidth;                    // assigning bandwidth

}
```

Chapter 3

Uniform Linear Array

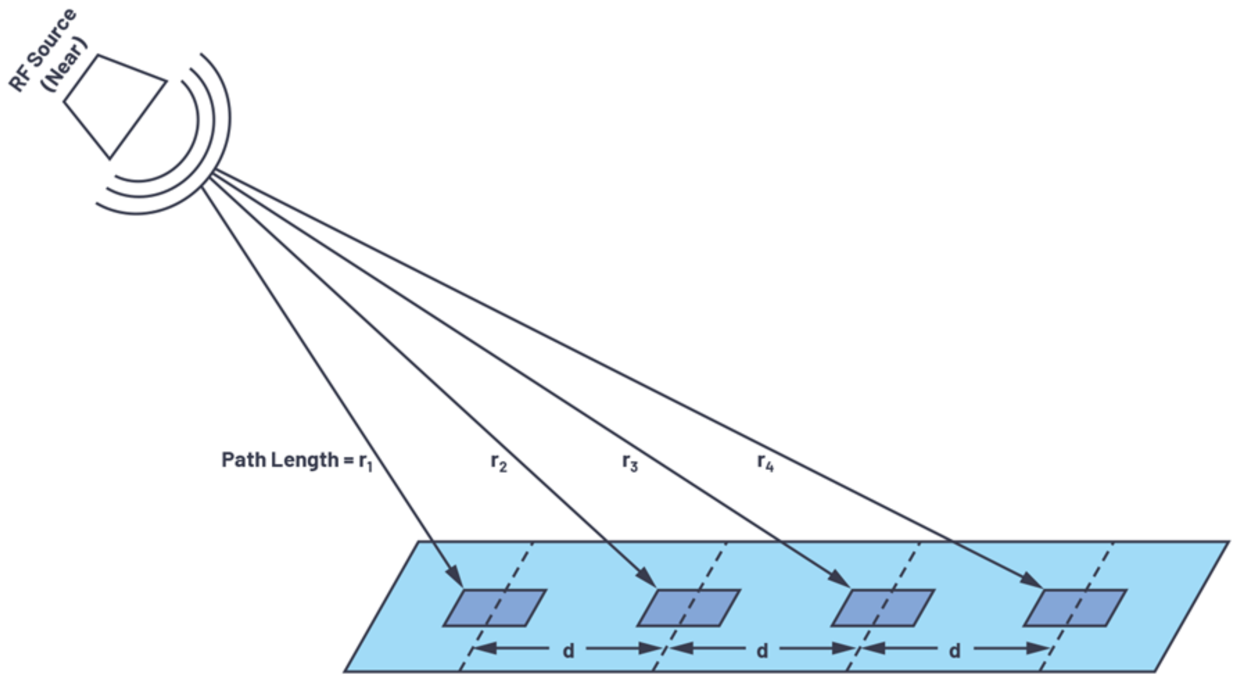


Figure 3.1: Uniform Linear Array

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;

    // 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
    int  frame_size;                 // the frame-size corresponding to a range cell in a decimated signal matrix

```

```
    std::vector<std::vector<complex<T>>> mulFFTMatrix; // the matrix containing the delays for each-element as a slot
    std::vector<complex<T>>             matchFilter;  // torch tensor containing the match-filter
    int    num_buffer_zeros_per_frame;                // number of zeros we're adding per frame to ensure no-rotation
    std::vector<std::vector<T>> beamformedImage;      // the beamformed image
    std::vector<std::vector<T>> cartesianImage;       // the cartesian version of beamformed image

    // Artificial acoustic-image related
    std::vector<std::vector<T>> currentArtificialAcousticImage; // acoustic image directly produced


    // Basic Constructor
    ULAClass() = default;

    // constructor
    ULAClass(const int    num_sensors_arg,
             const auto   inter_element_spacing_arg,
             const auto&  coordinates_arg,
             const auto&  sampling_frequency_arg,
             const auto&  recording_period_arg,
             const auto&  location_arg,
             const auto&  signalMatrix_arg,
             const auto&  lowpass_filter_coefficients_for_decimation_arg):
               num_sensors(num_sensors_arg),
               inter_element_spacing(inter_element_spacing_arg),
               coordinates(std::move(coordinates_arg)),
               sampling_frequency(sampling_frequency_arg),
               recording_period(recording_period_arg),
               location(std::move(location_arg)),
               signal_matrix(std::move(signalMatrix_arg)),
               lowpass_filter_coefficients_for_decimation(std::move(lowpass_filter_coefficients_for_decima
    {

        // calculating ULA direction
        sensor_direction = std::vector<T>{coordinates[1][0] - coordinates[0][0],
                                          coordinates[1][1] - coordinates[0][1],
                                          coordinates[1][2] - coordinates[0][2]};

        // normalizing
        auto    norm_value_temp   {std::norm(std::inner_product(sensor_direction.begin(),
                                                                sensor_direction.end(),
                                                                sensor_direction.begin(),
                                                                0.00))};

        // dividing
        if (norm_value_temp != 0) {sensor_direction =  sensor_direction / norm_value_temp;}

    }

    // // deleting copy constructor/assignment
    // ULAClass<T>(const ULAClass<T>& other)                 = delete;
    // ULAClass<T>& operator=(const ULAClass<T>& other)      = delete;
    ULAClass<T>(ULAClass<T>&& other)                     = delete;
    ULAClass<T>&   operator=(const ULAClass<T>& other)     = default;
```

```

    // member-functions
    void   buildCoordinatesBasedOnLocation();
    void   buildCoordinatesBasedOnLocation(const std::vector<T>& new_location);
    void   init(const TransmitterClass<T>&  transmitterObj);
    void   nfdc_CreateMatchFilter(const TransmitterClass<T>& transmitterObj);
    void   simulate_signals(const ScattererClass<T>&  seafloor,
                            const std::vector<std::size_t> scatterer_indices,
```

3.2 ULA Setup Scripts

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

```
template <typename T>
void fULASetup(ULAClass<T>&  ula_fls,
               ULAClass<T>&  ula_portside,
               ULAClass<T>&  ula_starboard)
{
    // setting up ula
    auto   num_sensors            {static_cast<int>(32)};         // number of sensors
    T      sampling_frequency     {static_cast<T>(160e3)};        // sampling frequency
    T      inter_element_spacing  {1500/(2*sampling_frequency)};  // space between
        samples
    T      recording_period       {10e-2};                        // sampling-period

    // building the direction for the sensors
    auto   ULA_direction          {std::vector<T>({-1, 0, 0})};
    auto   ULA_direction_norm     {norm(ULA_direction)};
    if (ULA_direction_norm != 0)  {ULA_direction = ULA_direction/ULA_direction_norm;}
    ULA_direction        =        ULA_direction  *  inter_element_spacing;

    // building coordinates for sensors
    auto   ULA_coordinates        {transpose(ULA_direction) * \
                                   linspace<double>(0.00,
                                                    num_sensors -1,
                                                    num_sensors)};

    // coefficients of decimation filter
    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;             //
        assigning number of sensors
    ula_fls.inter_element_spacing                = inter_element_spacing;   //
        assigning inter-element spacing
```

```
    ula_fls.coordinates                           = ULA_coordinates;          // assigning ULA coordinates
    ula_fls.sampling_frequency                    = sampling_frequency;       // assigning sampling frequencys
    ula_fls.recording_period                      = recording_period;         // assigning recording period
    ula_fls.sensor_direction                      = ULA_direction;            // ULA direction
    ula_fls.lowpass_filter_coefficients_for_decimation = lowpassfiltercoefficients; // storing coefficients


    // assigning values
    ula_portside.num_sensors                           = num_sensors;              // assigning number of sensors
    ula_portside.inter_element_spacing                 = inter_element_spacing;    // assigning inter-element spacing
    ula_portside.coordinates                           = ULA_coordinates;          // assigning ULA coordinates
    ula_portside.sampling_frequency                    = sampling_frequency;       // assigning sampling frequencys
    ula_portside.recording_period                      = recording_period;         // assigning recording period
    ula_portside.sensor_direction                      = ULA_direction;            // ULA direction
    ula_portside.lowpass_filter_coefficients_for_decimation = lowpassfiltercoefficients; // storing coefficients


    // assigning values
    ula_starboard.num_sensors                               = num_sensors;              // assigning number of sensors
    ula_starboard.inter_element_spacing                     = inter_element_spacing;    // assigning inter-element spacing
    ula_starboard.coordinates                               = ULA_coordinates;          // assigning ULA coordinates
    ula_starboard.sampling_frequency                        = sampling_frequency;       // assigning sampling frequencys
    ula_starboard.recording_period                          = recording_period;         // assigning recording period
    ula_starboard.sensor_direction                          = ULA_direction;            // ULA direction
    ula_starboard.lowpass_filter_coefficients_for_decimation = lowpassfiltercoefficients; // storing coefficients
}
```

Chapter 4

Autonomous Underwater Vehicle

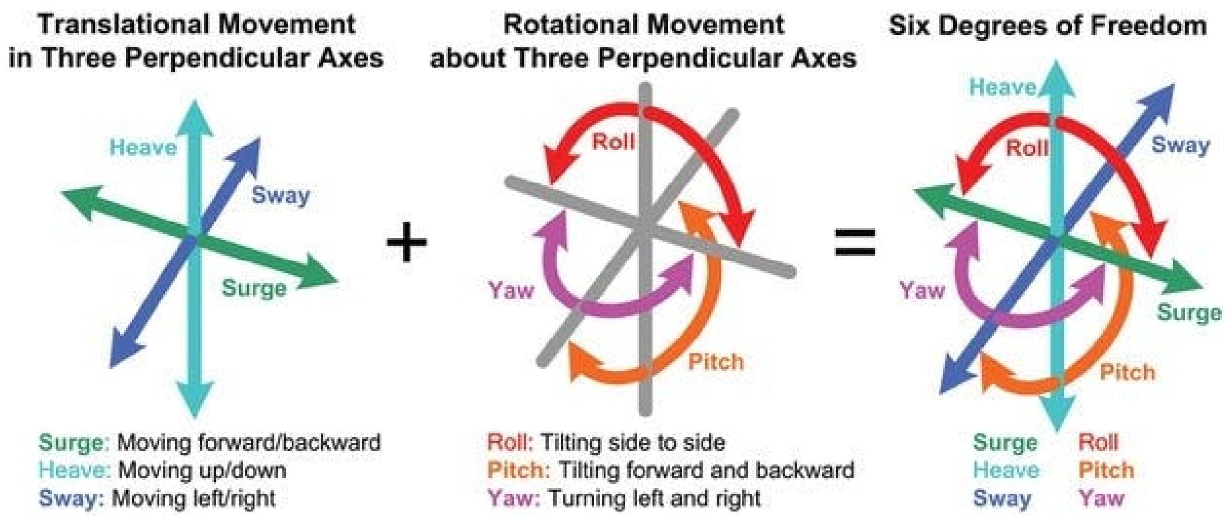


Figure 4.1: AUV degrees of freedom

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

```cpp
template <typename T>
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);
    void simulate_signal(const  ScattererClass<T>& seafloor,
                        svr::ThreadPool&            thread_pool);
    void subset_scatterers(const ScattererClass<T>& seafloor,
                          svr::ThreadPool&          thread_pool,
                          std::vector<std::size_t>& fls_scatterer_indices,
                          std::vector<std::size_t>& portside_scatterer_indices,
                          std::vector<std::size_t>& starboard_scatterer_indices);
    void step(T time_step);

};

/*============================================================================
Aim: update attributes
-----------------------------------------------------------------------------*/
template <typename T>
```

4.2 AUV Setup Scripts

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

```
template <typename T>
void fAUVSetup(AUVClass<T>& 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

}
```

Part II

Signal Simulation Pipeline

Part III

Imaging Pipeline

Part IV

Perception & Control Pipeline

Appendix A

General Purpose Templated Functions

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 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);
}
```

A.3 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){
```

```cpp
                        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);
}
```

A.4 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);
}
```

A.5 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);
    }
}
```

A.6 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};

        // calculating fft of two arrays
        auto   fft_A      {svr::fft(input_vector_A, canvas_length)};
        auto   fft_B      {svr::fft(input_vector_B, canvas_length)};

        // element-wise multiplying the two matrices
        auto   fft_AB     {fft_A *   fft_B};

        // finding inverse FFT
        auto   convolved_result  {ifft(fft_AB)};

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

    template <>
    auto   conv1D(const   std::vector<double>&   input_vector_A,
                  const   std::vector<double>&   input_vector_B)
    {
        // creating canvas
        auto   canvas_length     {input_vector_A.size() + input_vector_B.size() - 1};

        // calculating fft of two arrays
        auto   fft_A      {svr::fft(input_vector_A, canvas_length)};
```

```
    auto    fft_B      {svr::fft(input_vector_B, canvas_length)};

    // element-wise multiplying the two matrices
    auto    fft_AB     {fft_A *   fft_B};

    // finding inverse FFT
    auto    convolved_result  {ifft(fft_AB)};

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

/*==============================================================================
1D convolution of two vectors
> implemented through fft
------------------------------------------------------------------------------*/
template   <typename T1, typename T2>
auto    conv1D_fftw(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};

    // calculating fft of two arrays
    auto    fft_A      {svr::fft(input_vector_A, canvas_length)};
    auto    fft_B      {svr::fft(input_vector_B, canvas_length)};

    // element-wise multiplying the two matrices
    auto    fft_AB     {fft_A *   fft_B};

    // finding inverse FFT
    auto    convolved_result  {svr::ifft(fft_AB, fft_AB.size())};

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

/*==========================================================================
Long-signal Conv1D
improvements:
    > make an inplace version of this
--------------------------------------------------------------------------*/
template   <std::size_t   L,  typename   T>
auto    conv1D_long(const std::vector<T>&   input_vector_A,
                    const   std::vector<T>&   input_vector_B)
{
    // fetching dimensions
    const  auto    maxlength     {std::max(input_vector_A.size(),
                                           input_vector_B.size())};
    const  auto    filter_size    {std::min(input_vector_A.size(),
                                           input_vector_B.size())};
    const  auto    block_size     {L  +   filter_size - 1};
    const  auto    num_blocks     {2 + static_cast<std::size_t>(
        (maxlength - block_size)/L
    )};

```

```
        // obtaining references
        const  auto&  large_vector  {input_vector_A.size() >= input_vector_B.size() ? \
                                     input_vector_A       : input_vector_B};
        const  auto&  small_vector  {input_vector_A.size() < input_vector_B.size() ? \
                                     input_vector_A       :  input_vector_B};

        // setup
        auto   starting_index       {static_cast<std::size_t>(0)};
        auto   ending_index         {static_cast<std::size_t>(0)};
        auto   length_left_to_fill  {ending_index - starting_index};
        auto   canvas               {std::vector<double>(block_size, 0)};
        auto   finaloutput          {std::vector<double>(maxlength, 0)};
        auto   block_conv_output_size {L + 2 * filter_size -2};
        auto   block_conv_output    {std::vector<double>(block_conv_output_size, 0)};

        // block-wise processing
        for(auto   bid = 0; bid < num_blocks; ++bid)
        {
            // estimating indices
            starting_index       =   L*bid;
            ending_index         =   std::min(starting_index + block_size - 1,maxlength - 1);
            length_left_to_fill  =   ending_index - starting_index;

            // copying to the common-block
            std::copy(large_vector.begin() + starting_index,
                      large_vector.begin() + ending_index + 1,
                      canvas.begin());

            // performing convolution
            block_conv_output =   svr::conv1D_fftw(canvas,
                                                   small_vector);

            // discarding edges and writing values
            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);

    }

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

```
// 	// calculating fft of two arrays
// 	auto 	fft_A 		{svr::fft(input_vector_A, canvas_length)};
// 	auto 	fft_B 		{svr::fft(input_vector_B, canvas_length)};

// 	// element-wise multiplying the two matrices
// 	auto 	fft_AB 		{fft_A * 	fft_B};

// 	// finding inverse FFT
// 	auto 	convolved_result 	{ifft(fft_AB)};

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



/*==============================================================================
Short-conv1D
------------------------------------------------------------------------------*/
// template 	<std::size_t 	shortsize,
// 				typename 		T1,
// 				typename 		T2>
template 	<typename 		T1,
			 typename 		T2>
auto 	conv1D_short(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};

	// calculating fft of two arrays
	auto 	fft_A 		{svr::fft(input_vector_A, canvas_length)};
	auto 	fft_B 		{svr::fft(input_vector_B, canvas_length)};

	// element-wise multiplying the two matrices
	auto 	fft_AB 		{fft_A * 	fft_B};

	// finding inverse FFT
	auto 	convolved_result 	{ifft(fft_AB)};

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

}


/*==============================================================================
1D Convolution of a matrix and a vector
------------------------------------------------------------------------------*/
template 	<typename T>
auto 	conv1D(const 	std::vector<std::vector<T>>& input_matrix,
			   const 	std::vector<T>& 				input_vector,
			   const 	std::size_t& 					dim)
{
```

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

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

        // creating output based on dim
        if (dim == 1)
        {
            // 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);

    }

    /*==========================================================================
    1D Convolution of a matrix and a vector (in-place)
    --------------------------------------------------------------------------*/

}
```

A.7 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
    return std::move(spherical_vector);
}
/*==========================================================================
y = cart2sph(vector)
--------------------------------------------------------------------------*/
template <typename T>
auto   cart2sph_inplace(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 tesnor
    cartesian_vector[0]   =   azimuthal_angles;
    cartesian_vector[1]   =   elevation_angles;
    cartesian_vector[2]   =   rho_values;
}
/*==========================================================================
y = cart2sph(input_matrix, dim)
--------------------------------------------------------------------------*/
template <typename T>
auto cart2sph(const   std::vector<std::vector<T>>& input_matrix,
              const    std::size_t                  axis)
{
    // fetching dimensions
    const  auto&  num_rows   {input_matrix.size()};
    const  auto&  num_cols   {input_matrix[0].size()};

    // checking the axis and dimensions
    if (axis == 0 && num_rows != 3) {std::cerr << "cart2sph: incorrect num-elements
        \n";}
    if (axis == 1 && num_cols != 3) {std::cerr << "cart2sph: incorrect num-elements
        \n";}
```

```

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

    // if axis = 0, performing operation column-wise
    if(axis == 0)
    {
        for(auto col = 0; col < num_cols; ++col)
        {
            // fetching current column
            auto   curr_column   {std::vector<T>({input_matrix[0][col],
                                                  input_matrix[1][col],
                                                  input_matrix[2][col]})};

            // performing inplace transformation
            cart2sph_inplace(curr_column);

            // storing it back
            canvas[0][col]  =  curr_column[0];
            canvas[1][col]  =  curr_column[1];
            canvas[2][col]  =  curr_column[2];
        }
    }
    // if axis == 1, performing operations row-wise
    else if(axis == 0)
    {
        std::cerr << "cart2sph: yet to be implemented \n";
    }
    else
    {
        std::cerr << "cart2sph: yet to be implemented \n";
    }

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

}

// =====================================================================
template <typename T>
auto   sph2cart(const std::vector<T> spherical_vector){

    // creating cartesian vector
    auto   cartesian_vector  {std::vector<T>(spherical_vector.size(), 0)};

    // populating
    cartesian_vector[0]   =  spherical_vector[2] * \
                             cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
                             cos(spherical_vector[0] * std::numbers::pi / 180.00);
    cartesian_vector[1]   =  spherical_vector[2] * \
                             cos(spherical_vector[1] * std::numbers::pi / 180.00) * \
                             sin(spherical_vector[0] * std::numbers::pi / 180.00);
    cartesian_vector[2]   =  spherical_vector[2] * \
                             sin(spherical_vector[1] * std::numbers::pi / 180.00);

    // returning
```

```
        return std::move(cartesian_vector);
    }
}
```

A.8 Cosine

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

A.9 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) {}
};
```

A.10 Editing Index Values

```
#pragma once
/*==============================================================================
Matlab's equivalent of A[A < 0.5] = 0
------------------------------------------------------------------------------*/
// template <typename T, typename BooleanVector, typename U>
// auto edit(std::vector<T>&              input_vector,
//           BooleanVector                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
// }
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]};
        input_vector[target_index] += new_value;
    }

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

A.11 Equality

```
#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);
}
```

A.12 Exponentiate

```
#pragma once
```

```
/*==============================================================================
y = abs(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);
}
```

A.13 FFT

```
#pragma once
namespace svr {
    /*==============================================================================
    For type-deductions
    ------------------------------------------------------------------------------*/
    template <typename T>
    struct fft_result_type;

    // specializations
    template <> struct fft_result_type<double>{
        using type = std::complex<double>;
    };
    template <> struct fft_result_type<std::complex<double>>{
        using type = std::complex<double>;
    };
    template <> struct fft_result_type<float>{
        using type = std::complex<float>;
    };
    template <> struct fft_result_type<std::complex<float>>{
        using type = std::complex<float>;
    };

    template <typename T>
    using   fft_result_t   = typename fft_result_type<T>::type;

    /*==============================================================================
    y = fft(x, nfft)
        > calculating n-point dft where n-value is explicit
    ------------------------------------------------------------------------------*/
    template<typename T>
    auto fft(const    std::vector<T>&    input_vector,
             const    size_t             nfft)
    {
        // throwing an error
        if (nfft < input_vector.size()) {std::cerr << "size-mismatch\n";}
        if (nfft <= 0)                  {std::cerr << "size-mismatch\n";}
```

```

        // fetching data-type
        using   RType   = fft_result_t<T>;
        using   baseType   = std::conditional_t<std::is_same_v<T, std::complex<double>>,
                                                double,
                                                T>;

        // canvas instantiation
        std::vector<RType> canvas(nfft);
        auto    nfft_sqrt     {static_cast<RType>(std::sqrt(nfft))};
        auto    finaloutput   {std::vector<RType>(nfft, 0)};

        // calculating index by index
        for(int frequency_index = 0; frequency_index<nfft; ++frequency_index){
            RType  accumulate_value;
            for(int signal_index = 0; signal_index < input_vector.size(); ++signal_index){
                accumulate_value += \
                    static_cast<RType>(input_vector[signal_index]) * \
                    static_cast<RType>(std::exp(-1.00 * std::numbers::pi * \
                                                (static_cast<baseType>(frequency_index)/static_cast<baseType>(
                                                    * \
                                                static_cast<baseType>(signal_index)));
            }
            finaloutput[frequency_index] = accumulate_value / nfft_sqrt;
        }

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

    /*==========================================================================
    y = fft(std::vector<double> nfft) // specialization
    ------------------------------------------------------------------------*/
    #include <fftw3.h>       // for fft
    template <>
    auto fft(const std::vector<double>& input_vector,
             const std::size_t          nfft)
    {
        if (nfft < input_vector.size())
            throw std::runtime_error("nfft must be >= input_vector.size()");
        if (nfft <= 0)
            throw std::runtime_error("nfft must be > 0");

        // FFTW real-to-complex output
        std::vector<std::complex<double>> output(nfft);

        // Allocate input (double) and output (fftw_complex) arrays
        double* in = reinterpret_cast<double*>(fftw_malloc(sizeof(double) * nfft));
        fftw_complex* out =
            reinterpret_cast<fftw_complex*>(fftw_malloc(sizeof(fftw_complex) * (nfft/2 +
            1)));

        // Copy input and zero-pad if needed
        for (std::size_t i = 0; i < nfft; ++i) {
            in[i] = (i < input_vector.size()) ? input_vector[i] : 0.0;
        }

        // Create FFTW plan and execute
```

```
    fftw_plan plan = fftw_plan_dft_r2c_1d(static_cast<int>(nfft), in, out,
        FFTW_ESTIMATE);
    fftw_execute(plan);

    // Copy FFTW output to std::vector<std::complex<double>>
    for (std::size_t i = 0; i < nfft/2 + 1; ++i) {
        output[i] = std::complex<double>(out[i][0], out[i][1]);
    }
    // Optional: fill remaining bins with zeros to match full nfft size
    for (std::size_t i = nfft/2 + 1; i < nfft; ++i) {
        output[i] = std::complex<double>(0.0, 0.0);
    }

    // Cleanup
    fftw_destroy_plan(plan);
    fftw_free(in);
    fftw_free(out);

    // filling up the other half of the output
    const  auto   halfpoint  {static_cast<std::size_t>(nfft/2)};
    std::transform(
        output.begin() + 1,       // first half (skip DC)
        output.begin() + halfpoint, // end of first half
        output.rbegin(),       // start writing from last element backward (skip
            Nyquist)
        [](const auto& x) { return std::conj(x); }
    );

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


/*==============================================================================
y = ifft(x, nfft)
------------------------------------------------------------------------------*/
template<typename T>
auto ifft(const    std::vector<T>&   input_vector)
{
    // fetching data-type
    using  RType   = fft_result_t<T>;
    using  baseType   = std::conditional_t<std::is_same_v<T, std::complex<double>>,
                                        double,
                                        T>;

    //  setup
    auto   nfft       {input_vector.size()};

    // canvas instantiation
    std::vector<RType> canvas(nfft);
    auto   nfft_sqrt     {static_cast<RType>(std::sqrt(nfft))};
    auto   finaloutput   {std::vector<RType>(nfft, 0)};

    // calculating index by index
    for(int frequency_index = 0; frequency_index<nfft; ++frequency_index){
        RType  accumulate_value;
        for(int signal_index = 0; signal_index < input_vector.size(); ++signal_index){
            accumulate_value += \
                static_cast<RType>(input_vector[signal_index]) * \
```

```
                    static_cast<RType>(std::exp(1.00 * std::numbers::pi * \
                                             (static_cast<baseType>(frequency_index)/static_cast<baseType>(n
                                                 * \
                                             static_cast<baseType>(signal_index)));
            }
            finaloutput[frequency_index] = accumulate_value / nfft_sqrt;
        }

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

    /*==============================================================================
    y = ifft()
    using fftw
    ------------------------------------------------------------------------------*/
    // #include <fftw3.h>
    // #include <vector>
    // #include <complex>
    // #include <stdexcept>
    // #include <algorithm>

    // // template <typename T>
    // auto ifft(const std::vector<std::complex<double>>& input_vector,
    //           const std::size_t                        nfft)
    // {
    //     if (nfft <= 0)
    //         throw std::runtime_error("nfft must be > 0");
    //     if (input_vector.size() < nfft/2 + 1)
    //         throw std::runtime_error("input spectrum must have at least nfft/2+1
        bins");

    //     // Output: real-valued time-domain signal
    //     std::vector<double> output(nfft);

    //     // Allocate FFTW input/output
    //     fftw_complex* in =
        reinterpret_cast<fftw_complex*>(fftw_malloc(sizeof(fftw_complex) * (nfft/2 + 1)));
    //     double* out = reinterpret_cast<double*>(fftw_malloc(sizeof(double) * nfft));

    //     // Copy input spectrum into FFTW buffer
    //     for (std::size_t i = 0; i < nfft/2 + 1; ++i) {
    //         in[i][0] = input_vector[i].real();
    //         in[i][1] = input_vector[i].imag();
    //     }

    //     // Create inverse plan and execute
    //     fftw_plan plan = fftw_plan_dft_c2r_1d(static_cast<int>(nfft), in, out,
        FFTW_ESTIMATE);
    //     fftw_execute(plan);

    //     // Normalize by nfft (FFTW does unscaled IFFT)
    //     for (std::size_t i = 0; i < nfft; ++i) {
    //         output[i] = out[i] / static_cast<double>(nfft);
    //     }

    //     // Cleanup
    //     fftw_destroy_plan(plan);
    //     fftw_free(in);
```

```
    //    fftw_free(out);

    //    return output;
    // }

    /*========================================================================
    x = ifft(std::vector<std::complex<double>> spectrum, nfft)
    ------------------------------------------------------------------------*/
    #include <fftw3.h>
    #include <vector>
    #include <complex>
    #include <stdexcept>

    auto ifft(const std::vector<std::complex<double>>& input_vector,
              const std::size_t                    nfft)
    {
        if (nfft <= 0)
            throw std::runtime_error("nfft must be > 0");
        if (input_vector.size() != nfft)
            throw std::runtime_error("input spectrum must be of size nfft");

        // Output: real-valued time-domain sequence
        std::vector<double> output(nfft);

        // Allocate FFTW input/output
        fftw_complex* in = reinterpret_cast<fftw_complex*>(
            fftw_malloc(sizeof(fftw_complex) * (nfft/2 + 1)));
        double* out = reinterpret_cast<double*>(
            fftw_malloc(sizeof(double) * nfft));

        // Copy *only* the first nfft/2+1 bins (rest are redundant due to symmetry)
        for (std::size_t i = 0; i < nfft/2 + 1; ++i) {
            in[i][0] = input_vector[i].real();
            in[i][1] = input_vector[i].imag();
        }

        // Create inverse FFTW plan
        fftw_plan plan = fftw_plan_dft_c2r_1d(
            static_cast<int>(nfft), in, out, FFTW_ESTIMATE);

        fftw_execute(plan);

        // Normalize by nfft (FFTW leaves IFFT unscaled)
        for (std::size_t i = 0; i < nfft; ++i) {
            output[i] = out[i] / static_cast<double>(nfft);
        }

        // Cleanup
        fftw_destroy_plan(plan);
        fftw_free(in);
        fftw_free(out);

        return output;
    }



}
```

A.14 Flipping Containers

```
#pragma once
namespace svr {
    /*==============================================================================
    Mirror image of a vector
    ----------------------------------------------------------------------------*/
    template <typename T>
    auto    fliplr(const   std::vector<T>&    input_vector)
    {
        // creating canvas
        auto    canvas     {input_vector};

        // rewriting
        std::reverse(canvas.begin(), canvas.end());

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

A.15 Indexing

```
#pragma once
namespace svr {
    /*==========================================================================
    y = index(vector, mask)
    ------------------------------------------------------------------------*/
    template   <typename T1,
                typename T2,
                typename = std::enable_if_t<std::is_arithmetic_v<T1>            ||
                                            std::is_same_v<T1, std::complex<float> > ||
                                            std::is_same_v<T1, std::complex<double> >
                                           >
                >
    auto    index(const    std::vector<T1>&      input_vector,
                  const    std::vector<T2>&      indices_to_sample)
    {
        // creating canvas
        auto    canvas     {std::vector<T1>(indices_to_sample.size(), 0)};

        // copying the associated values
        for(int i = 0; i < indices_to_sample.size(); ++i){
            auto    source_index   {indices_to_sample[i]};
            if(source_index < input_vector.size()){
                canvas[i] = input_vector[source_index];
            }
            else
                cout << "svr::index | source_index !< input_vector.size()\n";
        }

        // returning
        return std::move(canvas);
    }
    /*==========================================================================
    y = index(matrix, mask, dim)
```

```
---------------------------------------------------------------------------*/
template <typename T1, typename T2>
auto index(const  std::vector<std::vector<T1>>& input_matrix,
           const  std::vector<T2>&             indices_to_sample,
           const  std::size_t&                 dim)
{
    // 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<T1>>()};

    // if indices are row-indices
    if (dim == 0){

        // initializing canvas
        canvas = std::vector<std::vector<T1>>(
            num_rows_matrix,
            std::vector<T1>(indices_to_sample.size())
        );

        // filling the canvas
        auto destination_index {0};
        std::for_each(indices_to_sample.begin(), indices_to_sample.end(),
                      [&](const auto& col){
                        for(auto row = 0; row < num_rows_matrix; ++row)
                            canvas[row][destination_index] =  input_matrix[row][col];
                        ++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);
}
```

```
}
```

A.16 Linspace

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


/*==============================================================================
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;}
};
/*==============================================================================
in-place
------------------------------------------------------------------------------*/
template <typename T>
auto linspace(std::vector<std::complex<T>>& 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 + static_cast<T>(i)*stepsize;
    }
};
/*==============================================================================
------------------------------------------------------------------------------*/
template <typename T>
auto linspace(const   T              startvalue,
             const    T              endvalue,
             const    std::size_t    numpoints)
{
    std::vector<T> input(numpoints);
    auto stepsize = static_cast<T>(endvalue - startvalue)/static_cast<T>(numpoints-1);
    for(int i = 0; i<input.size(); ++i) {input[i] = startvalue +
        static_cast<T>(i)*stepsize;}
    return std::move(input);
};
/*==============================================================================
------------------------------------------------------------------------------*/
template <typename T, typename U>
auto linspace(const   T              startvalue,
             const    U              endvalue,
```

```
            const    std::size_t    numpoints)
{
    std::vector<double> input(numpoints);
    auto stepsize = static_cast<double>(endvalue -
        startvalue)/static_cast<double>(numpoints-1);
    for(int i = 0; i<input.size(); ++i) {input[i] = startvalue + i*stepsize;}
    return std::move(input);
};
```

A.17 Max

```
#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);
}
```

A.18 Meshgrid

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


/*==============================================================================
mesh-grid when working with l-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});

}
```

A.19 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
```

```cpp
    return std::move(canvas);
}
```

A.20 Norm

```cpp
#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));
}
/*==============================================================================
------------------------------------------------------------------------------*/
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>
*/
```

A.21 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>
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){
        std::transform(input_matrix[row].begin(), input_matrix[row].end(),
                       canvas[row].begin(),
                       [&scalar](const auto& argx){
                           return argx/scalar;
                       });
    }

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

A.22 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];}

    // returning elements
    return a;
}
// =============================================================================
// y = matrix + matrix
template <typename T>
std::vector<std::vector<T>> operator+(const std::vector<std::vector<T>>& a,
                                      const std::vector<std::vector<T>>& b)
{
    // fetching dimensions
    const  auto&  num_rows_A    {a.size()};
    const  auto&  num_cols_A    {a[0].size()};
    const  auto&  num_rows_B    {b.size()};
    const  auto&  num_cols_B    {b[0].size()};

    // choosing the three different metrics
    if (num_rows_A != num_rows_B && num_cols_A != num_cols_B){
        cout << format("a.dimensions = [{},{}], b.shape = [{},{}]\n",
                       num_rows_A, num_cols_A,
                       num_rows_B, num_cols_B);
        std::cerr << "dimensions don't match\n";
    }

```

```
    // creating canvas
    auto   canvas    {std::vector<std::vector<T>>(
        std::max(num_rows_A, num_rows_B),
        std::vector<T>(std::max(num_cols_A, num_cols_B), (T)0.00)
    )};

    // performing addition
    if (num_rows_A == num_rows_B && num_cols_A == num_cols_B){
        for(auto row = 0; row < num_rows_A; ++row){
            std::transform(a[row].begin(), a[row].end(),
                           b[row].begin(),
                           canvas[row].begin(),
                           std::plus<T>());
        }
    }
    else if(num_rows_A == num_rows_B){

        // if number of columsn are different, check if one of the cols are one
        const  auto   min_num_cols  {std::min(num_cols_A, num_cols_B)};
        if (min_num_cols != 1) {std::cerr<< "Operator+: unable to broadcast\n";}
        const  auto   max_num_cols  {std::max(num_cols_A, num_cols_B)};

        // using references to tag em differently
        const  auto&  big_matrix    {num_cols_A > num_cols_B ? a : b};
        const  auto&  small_matrix  {num_cols_A < num_cols_B ? a : b};

        // Adding to canvas
        for(auto row = 0; row < canvas.size(); ++row){
            std::transform(big_matrix[row].begin(), big_matrix[row].end(),
                           canvas[row].begin(),
                           [&small_matrix,
                            &row](const auto& argx){
                               return argx + small_matrix[row][0];
                           });
        }
    }
    else if(num_cols_A == num_cols_B){

        // check if the smallest column-number is one
        const  auto   min_num_rows  {std::min(num_rows_A, num_rows_B)};
        if(min_num_rows != 1)    {std::cerr << "Operator+ : unable to broadcast\n";}
        const  auto   max_num_rows  {std::max(num_rows_A, num_rows_B)};

        // using references to differentiate the two matrices
        const  auto&  big_matrix    {num_rows_A > num_rows_B ? a : b};
        const  auto&  small_matrix  {num_rows_A < num_rows_B ? a : b};

        // adding to canvas
        for(auto row = 0; row < canvas.size(); ++row){
            std::transform(big_matrix[row].begin(), big_matrix[row].end(),
                           small_matrix[0].begin(),
                           canvas[row].begin(),
                           [](const auto& argx, const auto& argy){
                            return argx + argy;
                           });
        }
    }
    else {
        std::cerr << "operator+: yet to be implemented \n";
```

```
    }

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

    // adding scalar to the canvas
    std::transform(canvas.begin(), canvas.end(),
                   canvas.begin(),
                   [&scalar](auto& argx){return argx + scalar;});

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

    // adding scalar to the canvas
    std::transform(canvas.begin(), canvas.end(),
                   canvas.begin(),
                   [&scalar](auto& argx){return argx + scalar;});

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

A.23 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>>>>
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
    return std::move(canvas);
}
/*==============================================================================
y = vector * scalar
------------------------------------------------------------------------------*/
template <typename T>
auto    operator*(const   std::vector<T>&   input_vector,
                 const    T                 scalar)
{
    // creating canvas
    auto    canvas     {input_vector};
    // multiplying
    std::for_each(canvas.begin(), canvas.end(),
                 [&scalar](auto& argx){
                   argx = argx * scalar;
                 });
    // returning
    return std::move(canvas);
}
/*==============================================================================
y = vector * vector
------------------------------------------------------------------------------*/
template <typename T>
auto operator*(const std::vector<T>& input_vector_A,
               const std::vector<T>&  input_vector_B)
{
    // throwing error: size-desparity
    if (input_vector_A.size() != input_vector_B.size()) {std::cerr << "operator*: size
        disparity \n";}

    // creating canvas
    auto    canvas     {input_vector_A};

    // element-wise multiplying
    std::transform(input_vector_B.begin(), input_vector_B.end(),
                  canvas.begin(),
```

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

    // moving it back
    return std::move(canvas);
}
/*==============================================================================
------------------------------------------------------------------------------*/
template <typename T1, typename T2>
auto   operator*(const   std::vector<T1>&  input_vector_A,
                 const   std::vector<T2>&  input_vector_B)
{

    // checking size disparity
    if (input_vector_A.size() != input_vector_B.size())
        std::cerr << "operator*: error, size-disparity \n";

    // figuring out resulting data type
    using  T3   = decltype(std::declval<T1>() * std::declval<T2>());

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

    // performing multiplications
    std::transform(input_vector_A.begin(), input_vector_A.end(),
                  input_vector_B.begin(),
                  canvas.begin(),
                  [](const    auto&      argx,
                     const    auto&      argy){
                      return static_cast<T3>(argx) *  static_cast<T3>(argy);
                  });

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

}
/*==============================================================================
------------------------------------------------------------------------------*/
// scalar * matrix ==============================================================
template <typename T>
auto operator*(const T                          scalar,
               const std::vector<std::vector<T>>& inputMatrix)
{
    std::vector<std::vector<T>> temp {inputMatrix};
    for(int i = 0; i<inputMatrix.size(); ++i){
        std::transform(inputMatrix[i].begin(),
                      inputMatrix[i].end(),
                      temp[i].begin(),
                      [&scalar](T x){return scalar * x;});
    }
    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;
    }

    // case 3: when second matrix contains just one row
    // case 4: when first matrix is just one column
    // case 5: when second matrix is just one column

    // Otherwise, invalid
    else {
```

```
        throw std::runtime_error("operator* dimension mismatch");
    }
}
/*==============================================================================
y = scalar * matrix
------------------------------------------------------------------------------*/
template <typename T1, typename T2>
auto operator*(const T1                          scalar,
              const std::vector<std::vector<T2>>& inputMatrix)
{
    std::vector<std::vector<T2>> temp {inputMatrix};
    for(int i = 0; i<inputMatrix.size(); ++i){
        std::transform(inputMatrix[i].begin(),
                       inputMatrix[i].end(),
                       temp[i].begin(),
                       [&scalar](T2 x){return static_cast<T2>(scalar) * x;});
    }
    return temp;
}
/*==============================================================================
matrix-multiplication
------------------------------------------------------------------------------*/
template <typename T1, typename T2>
auto matmul(const std::vector<std::vector<T1>>& matA,
           const std::vector<std::vector<T2>>& matB)
{

    // throwing error
    if (matA[0].size() != matB.size())  {std::cerr << "dimension-mismatch \n";}

    // getting result-type
    using ResultType  = decltype(std::declval<T1>() * std::declval<T2>() + \
                                 std::declval<T1>() * std::declval<T2>() );

    // creating aliasses
    auto finalnumrows {matA.size()};
    auto finalnumcols {matB[0].size()};

    // creating placeholder
    auto rowcolproduct = [&](auto rowA, auto colB){
        ResultType temp {0};
        for(int i = 0; i < matA.size(); ++i) {temp +=
            static_cast<ResultType>(matA[rowA][i]) +
            static_cast<ResultType>(matB[i][colB]);}
        return temp;
    };

    // producing row-column combinations
    std::vector<std::vector<ResultType>> finaloutput(finalnumrows,
        std::vector<ResultType>(finalnumcols));
    for(int row = 0; row < finalnumrows; ++row){for(int col = 0; col < finalnumcols;
        ++col){finaloutput[row][col] = rowcolproduct(row, col);}}

    // returning
    return finaloutput;
}
/*==============================================================================
y = matrix * vector
------------------------------------------------------------------------------*/
```

```
template  <typename T>
auto operator*(const std::vector<std::vector<T>>  input_matrix,
               const   std::vector<T>               input_vector)
{
    // fetching dimensions
    const  auto&  num_rows_matrix   {input_matrix.size()};
    const  auto&  num_cols_matrix   {input_matrix[0].size()};
    const  auto&  num_rows_vector   {1};
    const  auto&  num_cols_vector   {input_vector.size()};

    const  auto&  max_num_rows      {num_rows_matrix > num_rows_vector ?\
                                     num_rows_matrix : num_rows_vector};
    const  auto&  max_num_cols      {num_cols_matrix > num_cols_vector ?\
                                     num_cols_matrix : num_cols_vector};

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

    //
    if (num_cols_matrix == 1 && num_rows_vector == 1){

        // writing to canvas
        for(auto   row = 0; row < max_num_rows; ++row)
            for(auto   col = 0; col < max_num_cols; ++col)
                canvas[row][col]   =   input_matrix[row][0]  *   input_vector[col];
    }
    else{
        std::cerr << "Operator*: [matrix, vector] | not implemented \n";
    }

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

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

A.24 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);
}
```

A.25 Operator Overloadings

A.26 Printing Containers

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

template<typename T>
void fpv(vector<T> input){
```

```
    for(auto x: input) cout << x << ",";
    cout << endl;
}
/*==============================================================================
------------------------------------------------------------------------------*/
template<typename T>
void fPrintMatrix(const std::vector<std::vector<T>> input_matrix){
    for(const auto& row: input_matrix)
        cout << format("{}\n", row);
}
/*==============================================================================
------------------------------------------------------------------------------*/
template <typename T>
void fPrintMatrix(const string&                input_string,
                  const std::vector<std::vector<T>> input_matrix){
    cout << format("{} = \n", input_string);
    for(const auto& row: input_matrix)
        cout << format("{}\n", row);
}
/*==============================================================================
------------------------------------------------------------------------------*/
template<typename T, typename T1>
void fPrintHashmap(unordered_map<T, T1> input){
    for(auto x: input){
        cout << format("[{},{}] | ", x.first, x.second);
    }
    cout <<endl;
}
/*==============================================================================
------------------------------------------------------------------------------*/
void fPrintBinaryTree(TreeNode* root){
    // sending it back
    if (root == nullptr) return;

    // printing
    PRINTLINE
    cout << "root->val = " << root->val << endl;

    // calling the children
    fPrintBinaryTree(root->left);
    fPrintBinaryTree(root->right);

    // returning
    return;

}
/*==============================================================================
------------------------------------------------------------------------------*/
void fPrintLinkedList(ListNode* root){
    if (root == nullptr) return;
    cout << root->val << " -> ";
    fPrintLinkedList(root->next);
    return;
}
/*==============================================================================
------------------------------------------------------------------------------*/
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());
}
```

A.27 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;
}
```

A.28 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);
}
/*==============================================================================
converting a matrix into a vector
------------------------------------------------------------------------------*/
```

```
template<typename T>
auto reshape(const std::vector<std::vector<T>>& input_matrix,
             const size_t                   M){

    // 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);
}
```

A.29 Summing with containers

```
#pragma once
/*==============================================================================
------------------------------------------------------------------------------*/
template <std::size_t axis, typename T>
auto sum(const std::vector<T>&  input_vector) -> std::enable_if_t<axis == 0,
    std::vector<T>>
{
    // returning the input as is
    return input_vector;
}
/*==============================================================================
------------------------------------------------------------------------------*/
template <std::size_t axis, typename T>
auto sum(const std::vector<std::vector<T>>& input_matrix) -> std::enable_if_t<axis == 0,
    std::vector<T>>
{
    // creating canvas
    auto   canvas    {std::vector<T>(input_matrix[0].size(), 0)};

    // filling up the canvas
    for(auto row = 0; row < input_matrix.size(); ++row)
        std::transform(input_matrix[row].begin(), input_matrix[row].end(),
                       canvas.begin(),
                       canvas.begin(),
                       [](auto& argx, auto& argy){return argx + argy;});

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

}
/*==============================================================================
------------------------------------------------------------------------------*/
template <std::size_t axis, typename T>
```

```cpp
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);
}
```

A.30 Tangent

```cpp
#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
```

```cpp
        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);
    }
}
```

A.31 Tiling Operations

```cpp
#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);
    }
}
```

A.32 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);
}
```

A.33 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";
```

```cpp

        // 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);
    }
}
```

A.34 Resetting Containers

```cpp
#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);

        // Recursively reset the remaining vectors
```

```
        if constexpr (sizeof...(rest_vectors) > 0) {
            reset(rest_vectors...);
        }
    }
}
```

A.35 Element-wise squaring

```
#pragma once
namespace svr {
    /*==============================================================================
    Element-wise squaring vector
    ------------------------------------------------------------------------------*/
    template   <typename T,
                typename = std::enable_if_t<std::is_arithmetic_v<T>>
               >
    auto   square(const  std::vector<T>& input_vector)
    {
        // creating canvas
        auto   canvas    {std::vector<T>(input_vector.size())};

        // peforming calculations
        std::transform(input_vector.begin(), input_vector.end(),
                       canvas.begin(),
                        [](const auto& argx){
                               return argx * argx;
                        });

        // moving it back
        return std::move(canvas);
    }
    /*==============================================================================
    Element-wise squaring vector (in-place)
    ------------------------------------------------------------------------------*/
    template   <typename T,
                typename = std::enable_if_t<std::is_arithmetic_v<T>>
               >
    void   square_inplace(std::vector<T>& input_vector)
    {
        // performing operations
        std::transform(input_vector.begin(), input_vector.end(),
                       input_vector.begin(),
                       [](auto& argx){
                           return argx * argx;
                       });
    }
    /*==============================================================================
    ELement-wise squaring a matrix
    ------------------------------------------------------------------------------*/
    template <typename T>
    auto   square(const  std::vector<std::vector<T>>& input_matrix)
    {
        // fetching dimensions
        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;}
}
```

A.36 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();
```

```cpp
            });

            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.37 Flooring

```cpp
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);
                           });
    }
}
```

A.38 Squeeze

```cpp
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);
    }
}
```

A.39 Tensor Initializations

```cpp
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);
    }
```

```
}
```

A.40 Real part

```
#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);
    }
    // /*==============================================================================
    // Element-wise real() of a matrix
    // ------------------------------------------------------------------------------*/
```

```
    // template  <typename  T>
    // auto   real(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)
    //    )};

    //    // 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(),
    //                       [](const auto& argx){
    //                           return std::real(argx);
    //                       });

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

A.41 Imaginary part

```
#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);
    }
}
```