A STL-based Header-Only Library for Scientific Computing

S.V. Rajendran

September 28, 2025

Preface

This “library” started off as a set of templated scripts developed to assist the AUV project. The AUV project originally started off with the aten/libtorch package, owing to the sheer amount of pre-built functions and classes for scientific computing. However, after much profiling and optimization, the latency of the pipeline didn’t reach the expected pipeline. I suspect the reason to be the compute-graphs thats built for backprop.

While there certainly are a number of other libraries that promise better efficiency, I took this opportunity to create my own. Not just to re-invent the wheel but also to practice template programming, which is something I’ve been wishing to brush-up on and master.

This library contains, primarily, templated functions and classes, designed for the AUV project. So, the presence of certain scientific functions, while the suspicious lack of others, should not be surprising. That being said, due to the sheer inter-disciplinary nature of AUV development, I would not be surprised if this library can also used for general-purpose scientific computing. And since the functions are written with standard STL algorithms and data-structures, readability is also made trivial.

Portability was one of priorities when setting out to build the AUV project. The svr library shares the same sentiment. Thus, svr is a header-only library, which, to grossly over-simplify it, means that you just need to unzip the contents of this library into your include-folder of your project, and include the main-header into your project. As someone who’s worked in domains ranging from naval warfare to surgical robotics, simple build systems is a priority. And thus, is a priority in all my projects.

To conclude this, already too long, preface, I would like to say that this is a library that I’m building in my spare time, to assist another project. Thus, not a first priority, or a second. I have a day-job as a Researcher at Computational Imaging & Systems Laboratory, and my second priority is building that AUV project. So the expectation is that the progress made from the other projects trickles down to this.

Contents

Chapter 1

Element-Wise Absolute Value

Overview

This function is primarily used to calculate the element-wise absolute-value of the argument. For scalar-containers, the return value is a container of same size, but with entries being the magnitude of their argument counter parts. For complex-containers, the return value is a container of same size, but with entries being the

$$\sqrt{x^2 + y^2}$$

where

- x is the real-component of the entry
- y is the imaginary-component of the entry

1.1 Usage

```cpp
#include "svr.hpp"

int main(){

    // absolute-values of a vector
    auto    input_vector   {std::vector<int>{-10, -5, 0, 5, 10}};
    auto    abs_input_vector  {svr::abs(input_vector)};

    // absolute values of a matri
    auto    input_matrix      {std::vector<std::vector<int>>{
        {-10,  -5,    0,  5,     10},
        {-20,  -10,   0,  20,    20},
        {-30,  -15,   0,  15,    30}
    }};
    auto    abs_input_matrix  {svr::abs(input_matrix)};

    // returning
    return 0;

}
```

1.2 Templated Implementation

```
// ===========================================================================
// 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);
}
```

Chapter 2

Boolean Comparators

Overview

A Boolean comparator is a logical mechanism that evaluates two or more inputs and produces an output based on Boolean conditions, returning either true or false. In general-purpose programming, Boolean comparators are implemented through relational operators such as `==, != , <, >, <=, and >=`, which compares the operands.

Comparators can also be combined using logical operators to form more complex conditions, enabling nuanced program flow. Here we show a number of templated operator overloads that implements the operations we need between the different classes of possible operands. In addition to templated overloads, we also provide concrete overloads to specify implementation for specific arguments.

2.1 Usage

Following are a few examples of using the operator

```cpp
#include "svr.hpp"

int main(){

    // input-configuration
    auto    input_vector          {std::vector<int>{-10, -5, 0, 5, 10}};

    // using
    auto    lessthan5             {input_vector < 5};
    cout << format("lessthan5                = {}\n", lessthan5);

    auto    lessthanorequal5      {input_vector <= 5};
    cout << format("lessthanorequal5         = {}\n", lessthanorequal5);

    auto    greaterthanminus5     {input_vector > -5};
    cout << format("greaterthanminus5        = {}\n", greaterthanminus5);

    auto    greaterthanorequaltominus5    {input_vector >= -5};
    cout << format("greaterthanorequaltominus5 = {}\n",
       greaterthanorequaltominus5);

    // returning
    return 0;

}
```

2.2 Source Code

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

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

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

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

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

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

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

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

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