

A STL-based Header-Only Library for Scientific Computing

S.V. Rajendran

September 30, 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 that’s 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 be 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 the 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 oversimplify 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

Preface	i
1 Element-Wise Absolute Value	1
1.1 Usage	1
1.2 Templated Implementation	2
2 Boolean Comparators	3
2.1 Usage	3
2.2 Source Code	4

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

```
1  #include "svr.hpp"
2
3  int main(){
4
5      // absolute-values of a vector
6      auto input_vector {std::vector<int>{-10, -5, 0, 5, 10}};
7      auto abs_input_vector {svr::abs(input_vector)};
8
9      // absolute values of a matrix
10     auto input_matrix {std::vector<std::vector<int>>{
11         {-10, -5, 0, 5, 10},
12         {-20, -10, 0, 20, 20},
13         {-30, -15, 0, 15, 30}
14     }};
15     auto abs_input_matrix {svr::abs(input_matrix)};
16
```

```

17     // returning
18     return 0;
19
20 }

```

1.2 Templated Implementation

```

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

```

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

```
1 #include "svr.hpp"
2
3 int main(){
4
5     // input-configuration
6     auto input_vector {std::vector<int>{-10, -5, 0, 5, 10}};
7
8     // using
9     auto lessthan5 {input_vector < 5};
10    cout << format("lessthan5 = {}\n", lessthan5);
11
12    auto lessthanoqual5 {input_vector <= 5};
13    cout << format("lessthanoqual5 = {}\n", lessthanoqual5);
14
15    auto greaterthanminus5 {input_vector > -5};
16    cout << format("greaterthanminus5 = {}\n", greaterthanminus5);
17
```

```

18     auto    greaterthanorequaltominus5    {input_vector >= -5};
19     cout << format("greaterthanorequaltominus5 = {}\n",
20                     greaterthanorequaltominus5);
21
22     // returning
23     return 0;
24 }

```

2.2 Source Code

```

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

```

```

40 // =====
41 template <typename T, typename U>
42 auto operator>(const std::vector<T>&    input_vector,
43               const U                  scalar)
44 {
45     // creating canvas
46     auto canvas {std::vector<bool>(input_vector.size())};
47
48     // transforming
49     std::transform(input_vector.begin(), input_vector.end(),
50                   canvas.begin(),
51                   [&scalar](const auto& argx){
52                       return argx > static_cast<T>(scalar);
53                   });
54
55     // returning
56     return std::move(canvas);
57 }
58 /*=====
59 -----*/
60 template <typename T, typename U>
61 auto operator>=(const std::vector<T>&    input_vector,
62               const U                  scalar)
63 {
64     // creating canvas
65     auto canvas {std::vector<bool>(input_vector.size())};
66
67     // transforming
68     std::transform(input_vector.begin(), input_vector.end(),
69                   canvas.begin(),
70                   [&scalar](const auto& argx){
71                       return argx >= static_cast<T>(scalar);
72                   });
73
74     // returning
75     return std::move(canvas);
76 }

```
