

IMPLEMENTING SYMBOLIC MATHEMATICS IN A STATICALLY-TYPED LANGUAGE

SymboMath Symbolic Mathematics in C++

Toby Davis

Hampton School

August 13, 2022

Abstract

While computers excel at solving numerical calculations, it's incredibly difficult to efficiently implement symbolic calculations – i.e., equations including variables, derivatives and integrals. For numeric calculations and the vast majority of software, symbolic mathematics provides very limited benefits, however in certain applications, such as visual calculators and scientific programs, the ability to manipulate equations algebraically can be incredibly useful.

This paper investigates how symbolic mathematics can be implemented in C++ , an object-oriented, statically-typed language which operates closer to the hardware than languages like Python and Java.

Summary

1	What is Symbolic Mathematics?	4
1.1	Why is it Useful?	4
1.2	Why C++ ?	5
1.3	What About Automatic Differentiation?	5
2	SymboMath's Goals	7
2.1	Numeric Evaluation	7
2.2	Functions	7
2.3	Variables	7
2.4	Calculus	7
2.5	Simplification	8
3	Existing Symbolic Mathematics Libraries	9
4	Overall Program Design	10
4.1	Data Types	10
4.1.1	Examples	11
4.2	Algorithms	11
4.2.1	Example	12
4.3	External Libraries	13
5	Implementing the Data Types	14
5.1	The Component Type	14
5.1.1	Evaluation	14
5.2	Type Detection	14
5.3	Naming	14
5.4	Helpers	15
5.5	The Number Type	15
5.6	Variables	15
5.7	Functions	16
5.7.1	The Functor	16
5.7.2	Functor Arguments	16
5.7.3	Function Evaluation	16
5.8	The Tree Type	17

6	Processing the Equation Input	19
6.1	Tokenizing	19
6.1.1	Example	19
6.2	Lexing	20
6.3	Processing the Lexed Results	21

1 What is Symbolic Mathematics?

Symbolic mathematics is “the use of computers to manipulate mathematical equations and expressions in symbolic form, as opposed to manipulating the numerical quantities represented by those symbols.” [1] In practice, this means the computer can deal with an expression like $5x + 7$, instead of just $5 \cdot 2 + 7$. Additionally, the computer would be able to tell you that

$$\frac{d}{dx} 2x^2 = 4x \tag{1}$$

instead of evaluating/approximating the derivative at a given point by using the definition for the derivative (or similar):

$$\frac{d}{dx} f(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \tag{2}$$

In addition, it is possible to integrate certain equations symbolically to obtain exact results, as opposed to being constrained to definite integration using Riemann Sums or other approximation methods.

1.1 Why is it Useful?

The most common use-case for a symbolic maths is for solving incredibly large, tedious problems, which would be prone to trivial errors if solved by a human [2]. By providing a computer with instructions and operations to manipulate and solve equations step-by-step, it’s possible to answer questions that had been ignored for millennia due to their scale and complexity.

Another common use for symbolic maths engine is to present data in a more human-readable form. For example, **WolframAlpha** is capable of taking complex, multi-variable equations as inputs and manipulating them to solve equations, plot interactive graphs and provide general solutions to queries.

As mentioned earlier, symbolic maths engines can also directly integrate or differentiate an expression, providing exact answers as opposed to approximations. The approximations generated from naive methods are often slow to compute (especially for integration), and require precise calculations and, in some cases, vast numbers of iterations to converge on an accurate value. A symbolic expression, however, can be evaluated very efficiently and, as there are no approximations present in the integration/differentiation, give exact results.

1.2 Why C++ ?

C++ is a statically-typed language based on C, providing classes, structs, template-meta-programming and more. It is also possible to write highly efficient code in C++ , making it the programming language of choice for many high-performance applications.

Being statically typed, using C++ adds another layer of complexity to implementing a symbolic maths library, as it makes it vastly more complicated to deal with different types for numbers, variables, functions, operations, etc.

If this project were to be implemented in Python, which is dynamically typed and hence does not care about data types, it would be much easier to design a data-structure which could store a symbolic equation and operate on it accordingly.

Ultimately, I chose C++ for this project as it would pose more of a challenge, as well as allowing more room for optimisation of performance and efficiency. C++ is also the language I am most comfortable in using, and, fortunately, appears to have a limited number of fully-featured symbolic math libraries available for it [3].

1.3 What About Automatic Differentiation?

When researching symbolic mathematics, particularly symbolic *differentiation*, one is bound to come across the term “automatic differentiation”. This varies from symbolic mathematics in a few significant ways.

Firstly, automatic differentiation does not operate on symbolic equations. It operates entirely numerically, though may operate on vectors, matrices or arrays – not just scalar values.

Secondly, automatic differentiation aims to provide a means of computing a derivative of a function at a given point, not in the general case. This may seem exactly like the approximation methods spoken about earlier, but it does still compute a function for the derivative (just in a very different manor).

Automatic differentiation is most commonly used in machine-learning applications, where a complex activation function needs to be differentiated in order to make use of the gradient descent algorithm. The reason it’s used here is because, given a static, compile-time definition for a function, it’s possible to generate another function at compile-time which represents the derivative of the initial function, meaning no time needs to be spent at

runtime parsing the input and differentiating it before evaluating the result.

2 SymboMath's Goals

`SymboMath`¹ should be a library usable in any CMake project, and should be wrapped in an easy-to-use, C++-styled class for easy reuse of variables, equations and functions.

`SymboMath` *could* also support some form of basic equation solving, though this might be outside the scope of simple algebraic manipulation of equations.

2.1 Numeric Evaluation

The program should be able to evaluate any numeric expression involving the four basic arithmetic operations, as well as exponentiation, trigonometry and roots.

2.2 Functions

It should be easy to register custom functions with procedures to evaluate, differentiate and potentially integrate them. There should also be a wide set of functions already implemented, including trigonometry, logarithms and exponents, square roots, etc.

2.3 Variables

`SymboMath` must be able to support user-defined variables, including value substitution, manipulation of equations including variables and potentially simplification of expressions containing variables (see 2.5). Variables should act like any other object within the data structure of an equation, and operations applied to them should perform as expected, with no unwanted side-effects.

2.4 Calculus

The primary data structure used in `SymboMath` should be easily differentiated given that every function being used has a well-defined derivative. Integration of functions is more complex, but should be possible in most simple cases, and some more advanced scenarios. Calculus should also be performed with respect to a specified variable, meaning multi-variable equations can still be

¹`SymboMath` GitHub repository: <https://github.com/pencilcaseman/symbomath>

operated on correctly. In such cases, all other variables will be treated as constants.

2.5 Simplification

Ideally, **SymboMath** should be able to simplify a given expression and reduce it to its most fundamental form. This means $2 \cdot 5x$ will automatically be transformed into $10x$, and $\frac{10}{2}x$ will be transformed into $5x$. This will be most important in the result of differentiation/integration, as, depending on how these are implemented, the resulting expressions may include many redundant operations and values.

Simplification is likely to be the most difficult part of the **SymboMath** project and hence will be implemented as the last step. The equations returned by the program will/must still be valid regardless of whether they've been simplified or not, so the program can still be verified without this step.

3 Existing Symbolic Mathematics Libraries

Pre-existing symbolic mathematics libraries are a good place to start to identify implementation techniques and overall design architectures, as they are shown to work effectively.

`mathiu.cpp` is a C++ symbolic maths library based heavily on pattern matching. By making use of the `match(it)` library, `mathiu.cpp` can combine and compare operations and expressions to form an equation. Differentiation is performed recursively, applying a small set of rules to individual functions and combining them to produce the final result.

Wolfram Mathematica is another program capable of symbolic manipulation of equations. Although Mathematica's code is not available online, there are detailed explanations of how it's internal systems handle the data on the Wolfram Reference website [5]. According to their explanations, they have four fundamental data types:

Numbers

Strings

Symbols

General Expressions

Everything in the Wolfram Language keeps a reference count, tracking how many other Wolfram Language objects are pointing to it. This allows the language to free anything immediately after the last thing pointing to it is destroyed, saving memory and improving efficiency.

To process the inputs provided to the language, the Wolfram Language first creates an expression representing the input. It then applies all known rules for the contained objects before outputting the result. The important thing to note here is that *all* known rules are applied. This implies that no single rule can achieve the desired result, and that intelligently analysing the expression to identify the most effective rule is not always possible.

4 Overall Program Design

4.1 Data Types

Based on previous research and extensive testing, a polymorphic approach using C++ pointers and virtual functions appears to be the most efficient way of implementing the required features. Additionally, instead of using raw pointers, the program will make use of the C++ standard library's `std::shared_ptr<T>` type, as it can automatically handle reference counting and memory allocation/deallocation, reducing the scope for memory leaks and bugs in the final program.

Another benefit of using `std::shared_ptr<T>` is that, if all classes inherit from a single base type, it's possible to cast between the base type and the inherited types at runtime. This will allow for the creation of trees, but with no branching or data type requirements, which will be elaborated on later.

`SymboMath` will make use of the following data types:

`Component`: The most fundamental type in the system
`Number`: Stores a single numeric value
`Variable`: Stores a string value representing a variable name
`Function`: An operation taking n arguments, returning a scalar
`Tree`: Stores an equation and provides some utility functions

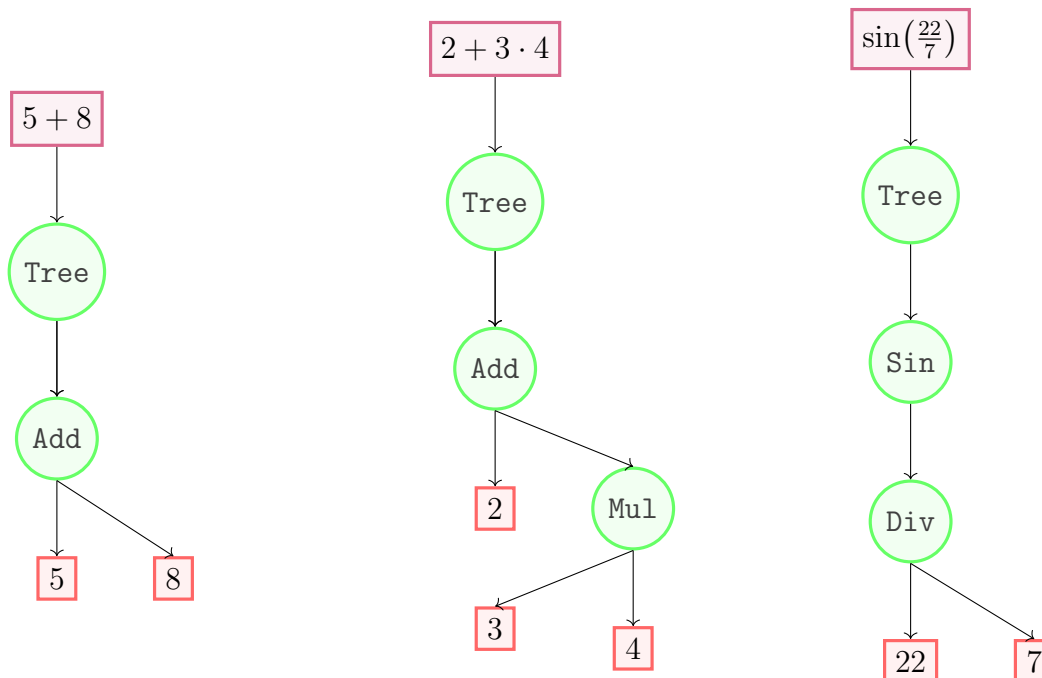
All objects will store only the data and instructions necessary for their operation, and will not access global variables unless it is necessary (searching for a function, for example).

An equation will be a `Tree` object storing a single `Function` object, which will be the highest-level function/operation in the equation. The function objects will have n branches, where n represents the number of arguments they take. For an arithmetic operation such as addition or subtraction, $n = 2$. For a single-argument function like $\sin(x)$, $n = 1$.

Each branch of the function will contain yet another object – this may be a `Number`, `Variable` or a `Function`. Thankfully, the data structure identified

earlier (elaborated on later) allows for any of these types to be stored, effectively circumventing the C++ static-typing system which makes this problem hard in the first place.

4.1.1 Examples



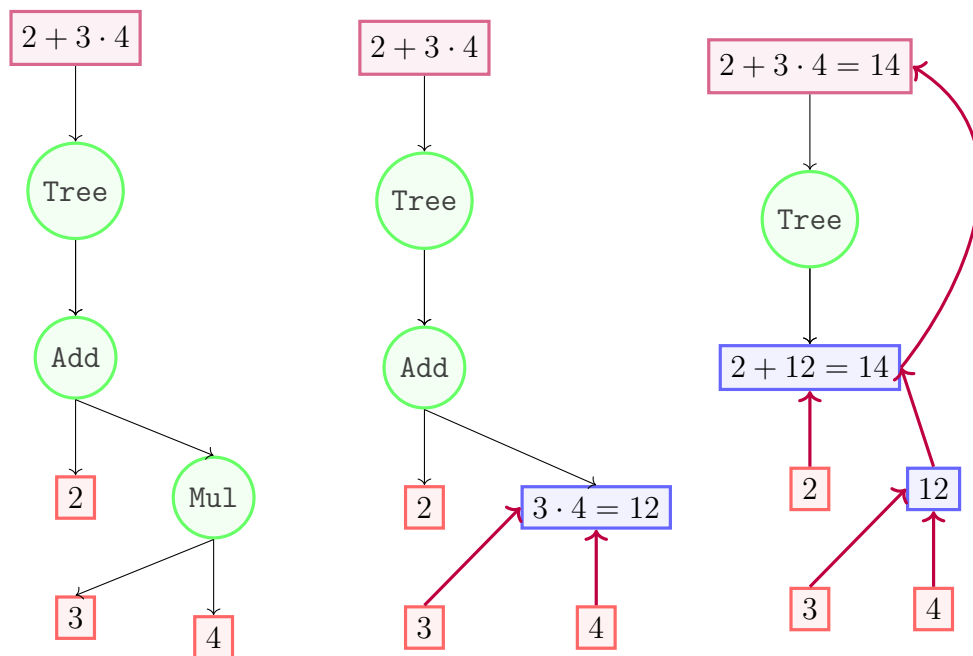
4.2 Algorithms

All high-level algorithms in **SymboMath** will operate recursively on the data they contain, and should not require external information to operate correctly.

For example, to evaluate the numeric result of an expression, the `eval` function would be called on the `Tree` object containing the equation. The `Tree` will then call `eval` on the primary function/operation, which, in turn, calls `eval` on each of its branches. The numeric result of these function calls can then be propagated back up the tree, applying the relevant operations where necessary. This results in a final numeric result being returned from the `Tree` object.

4.2.1 Example

For example, let's evaluate the middle example from 4.1.1.



Hopefully, the example above shows that it's possible to evaluate a tree recursively, from the bottom up, to calculate a numeric result. These steps can be extended to differentiation, simplification and potentially integration.

4.3 External Libraries

In order to reduce the chance of bugs appearing and to improve functionality, SymboMath makes use of a few external libraries.

```
SymboMath
├── LibRapid [6] -- Provides advanced maths functionality
│   ├── MPFR [7] -- Support for multi-precision floating point types
│   └── MPIR [8] -- Required by MPFR
├── {fmt} [9] -- For faster, easier string manipulation and printing
└── scnlib [10] -- Provides fast string scanning and casting routines
```

5 Implementing the Data Types

5.1 The Component Type

For the data structure identified in 4.1, it's necessary to have a single base object which all other types will extend. This data type should have very minimal functionality, and all the functions it provides should ultimately be overwritten.

Internally, the Component type contains virtual functions which throw an error when called, ensuring they cannot be called accidentally and cause unwanted results.

To start with, none of the classes will support any advanced features like simplification or calculus, but will support a simple, numeric evaluation function so we can check that the whole system is working correctly.

```
1 virtual Scalar eval() const;  
2 virtual std::string type() const;  
3 virtual std::string name() const;  
4 virtual std::string str(uint64_t indent) const;  
5 virtual std::string repr(uint64_t indent, uint64_t typeWidth,  
6                          uint64_t valWidth) const;
```

5.1.1 Evaluation

The eval function returns a scalar value, which is the result of evaluating the object. The evaluation function for each type will be covered in more detail later on.

5.2 Type Detection

The type function returns the type of the object (i.e. "NUMBER", "VARIABLE", "FUNCTION", etc.). This is used in the cases where more information needs to be known about a specific object, as the program cannot apply a general algorithm. It's also useful for debugging purposes, and is closely linked to the helper functions.

5.3 Naming

Objects like functions and variables have names associated with them, and hence a procedure to extract the name of a given object is required. For

objects which do not have a name associated with them, calling this function will raise an error, preventing unwanted results.

5.4 Helpers

The `str` and `repr` functions are purely to debug the algorithms and ensure that everything is working correctly. They both return string representations of the object, but in slightly different formats, showing different information.

5.5 The Number Type

The `Number` class is incredibly simple, and stores a single `Scalar` object (this is a type definition to allow different types to be used). The `eval` function simply returns the value stored in the class.

```
1 class Number : public Component {
2 public:
3     /* [ code omitted ] */
4
5     Scalar eval() const override { return m_value; }
6
7     /* [ code omitted ] */
8 private:
9     Scalar m_value = 0;
10 };
```

5.6 Variables

Variables such as x and y cannot be evaluated numerically without some form of substitution, so for now, the class will act as a stub which can be expanded on later as more features are added.

It is still important to implement the `name` function, however, as this will be needed for variable look up further down the line.

```
1 class Variable : public Component {
2 public:
3     /* [ code omitted ] */
4
5     std::string name() const override { return m_name; }
6
7     /* [ code omitted ] */
8 private:
```

```

9     std::string m_name = "NONAME";
10 };

```

5.7 Functions

The Function class is responsible for handing arithmetic operations such as addition and subtraction and single-argument functions like $\sin(x)$ and $\cos(y)$. The type can also be used to handle multi-argument functions like $\max(x, y)$ if required.

5.7.1 The Functor

In order to maintain runtime-compatible evaluation and to reduce the complexity of the program, the functor member of the function objects will be an instance of the C++ standard library function type, which provides a simple way to wrap a lambda function.

The functor will take a list of Scalar objects and will return another Scalar object, which is the result of the calculation.

5.7.2 Functor Arguments

The Function class also needs to know what arguments it's operating on, and these must be set at runtime. For this reason, each instance of the class will store a list of Component pointers (remember, these could be numbers, variables or functions) – one for each argument to the function.

One downside to this approach is that the number of arguments cannot be determined from the functor argument itself, so another member variable is required to store this information, with a function to access it.

5.7.3 Function Evaluation

In order to evaluate the result of the function, the arguments, which are still Component pointers, must first be evaluated and stored in a list. That list can then be fed directly into the functor object and the result can be returned.

```

1 class Function : public Component {
2 public:
3     /* [ code omitted ] */

```



```

4
5 Scalar eval() const override {
6     std::vector<Scalar> operands;
7     for (const auto &val : m_values)
8         operands.push_back(val->eval());
9     return m_functor(operands);
10 }
11
12 uint64_t numOperands() const { return m_numOperands; }
13
14 std::string name() const override { return m_name; }
15
16 /* [ code omitted ] */
17
18 private:
19     std::string m_name = "NULLOP";
20     std::function<Scalar(const std::vector<Scalar> &)>
21         m_functor;
22     uint64_t m_numOperands = 0;
23
24     std::vector<std::shared_ptr<Component>> m_values = {};
25 };

```

5.8 The Tree Type

The Tree class is incredibly simple, mainly intended to store a single function object which can then be evaluated. It also provides a few helper functions to format equations more nicely, though these are purely visual and have no impact on the program itself.

In the code listing, you may notice that the `m_tree` member is a list of pointers; this is done for future-proofing to allow variable values to be stored on a per-equation basis, for example. Currently it serves no purpose and can easily be changed to store a single object.

```

1 class Tree : public Component {
2 public:
3     /* [ code omitted ] */
4
5     Scalar eval() const override { return m_tree[0]->eval(); }
6
7     /* [ code omitted ] */
8 private:
9     std::vector<std::shared_ptr<Component>> m_tree;

```

¹⁰ };



6 Processing the Equation Input

Before it's possible to generate the tree from the input equation (in this case, a `std::string`), the text must first be processed into a form that can be understood more easily by the computer.

6.1 Tokenizing

The first step in transforming the user's input into something the computer can process is called “tokenizing”². It's the process of splitting the input string into a list of “tokens”, which, in this case, are individual characters³ the program is allowed to use.

The process is very simple, and involves iterating over each character in the input text, checking whether it's a valid character to be in an equation (e.g. emojis are invalid) and, if it is, storing it in a list. This process also removes spaces from the input, as it's not important to the equation itself.

Another data type is required to store the result of this process, which is very simple and contains information about the type of the value stored, along with the actual character.

```
1 struct Token {  
2     uint64_t type;  
3     char val;  
4 };
```

Listing 1: The token object definition

6.1.1 Example

For example, the input $41 + 1$ will produce the list of tokens shown below. Note, however, that `TYPE_DIGIT`, `TYPE_ADD` and `TYPE_OPERATION` are all numeric values.

```
1 // ===== [ 0 ] =====  
2 {  
3     uint64_t type = TYPE_DIGIT;  
4     char val     = '4';  
5 }
```

²Tokenizing and lexing are technically the same process, however, in `SymboMath` the process is split into two parts, and “tokenizing” fits the first part more closely than “lexing”

³Character as in an ASCII character – not just letters of the alphabet

```

6 // ===== [ 1 ] =====
7 {
8     uint64_t type = TYPE_DIGIT;
9     char val      = '1';
10 }
11 // ===== [ 2 ] =====
12 {
13     uint64_t type = TYPE_ADD | TYPE_OPERATION;
14     char val      = '+';
15 }
16 // ===== [ 3 ] =====
17 {
18     uint64_t type = TYPE_DIGIT;
19     char val      = '1';
20 }

```

6.2 Lexing

The individual characters returned from the tokenizing step are entirely useless unless we can make sense of what they mean in a larger context. This is the process of the “lexer”, which takes the characters identified in the previous step and produces a list of objects which the computer can easily make sense of.

The algorithm itself is quite simple, and joins together strings of digits to form numbers, strings of characters to form text and identifies arithmetic operations and brackets.

Another data type is also required for the output of the lexing process. It’s almost exactly the same as the token object, except it stores a string instead of a single character.

Taking the example given in 6.1.1, the algorithm will output the following.

```

1 // ===== [ 0 ] =====
2 {
3     uint64_t type    = TYPE_NUMBER | TYPE_VARIABLE;
4     std::string val = "41";
5 }
6 // ===== [ 1 ] =====
7 {
8     uint64_t type    = TYPE_ADD | TYPE_OPERATION;
9     std::string val = "+";
10 }
11 // ===== [ 2 ] =====

```

```
12 {  
13     uint64_t type    = TYPE_NUMBER | TYPE_VARIABLE;  
14     std::string val = "1";  
15 }
```

This example is incredibly simple, and could be achieved simply by splitting the equation by spaces, for example. In more complex situations, however, such a technique will not work and may lead to unwanted edge-case scenarios – this method avoids those completely by interpreting the input character by character.

6.3 Processing the Lexed Results

This section of the program isn't necessarily required, but it will serve a very important purpose later down the line, as well as allowing for a few more mathematical inputs.

When writing an equation by hand, it's not uncommon to write something like $5(3x-2)$ or $2\sin(x)$. In these cases, we understand that the term preceded by a coefficient should be multiplied by that coefficient, however the computer doesn't understand this.

In order to simplify things, we need to insert a `Lexed{TYPE_MUL, "*"}` between the coefficient and the rest of the term.

References

- [1] TheFreeDictionary.com. (n.d.). symbolic mathematics. [online] Available at: <https://encyclopedia2.thefreedictionary.com/symbolic+mathematics> [Accessed 9 Aug. 2022].
- [2] Clapp, L.C. and Kain, R.Y. (1963). A computer aid for symbolic mathematics. Proceedings of the November 12-14, 1963, fall joint computer conference on XX - AFIPS '63 (Fall). doi:10.1145/1463822.1463877.
- [3] GitHub.com (n.d.). Build software better, together. [online] Available at: <https://github.com/topics/symbolic-math?l=c%2B%2B> [Accessed 9 Aug. 2022].
- [4] Fu, B. (2022). mathiu.cpp. [online] GitHub. Available at: <https://github.com/BowenFu/mathiu.cpp> [Accessed 10 Aug. 2022].
- [5] reference.wolfram.com. (n.d.). The Internals of the Wolfram System—Wolfram Language Documentation. [online] Available at: <https://reference.wolfram.com/language/tutorial/TheInternalsOfTheWolframSystem.html> [Accessed 10 Aug. 2022].
- [6] Davis, T. (2022). LibRapid: Optimised Mathematics for C++ (Version 0.5.0) [Computer software]. <https://github.com/LibRapid/librapid>
- [7] Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P. and Zimmermann, P. (2007). MPFR. ACM Transactions on Mathematical Software, 33(2), p.13. doi:10.1145/1236463.1236468.
- [8] B. Gladman, W. Hart, J. Moxham, et al. MPIR: Multiple Precision Integers and Rationals, 2015. version 2.7.0, A fork of the GNU MP package (T. Granlund et al.) <http://mpir.org>.
- [9] fmtlib (2019). fmtlib/fmt. [online] GitHub. Available at: <https://github.com/fmtlib/fmt>.
- [10] Kosunen, E. (2022). scnlib. [online] GitHub. Available at: <https://github.com/eliaskosunen/scnlib> [Accessed 11 Aug. 2022].