Implementing Symbolic Mathematics in a Statically-Typed Language

SymboMath
Symbolic Mathematics in C++

Toby Davis

Hampton School

August 10, 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?

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 $5 \times 2 + 7$. 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):

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

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 How Does it Differ from Automatic Differentiation?

When researching symbolic mathematics, 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 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.

SymboMath's code can be found online at:
https://github.com/pencilcaseman/symbomath

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 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 it's most fundamental form. This means $2 \times 5x$ 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

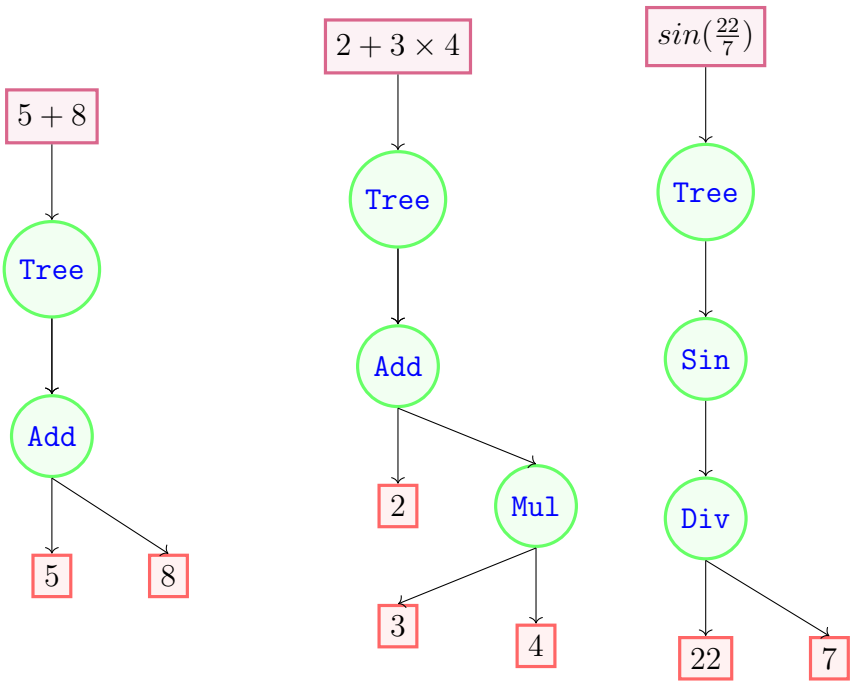
All objects will store only the data and instructions necessary for their operation, and will not access global variables unless it is necessary (searching for a function, for example).

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. For example,

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].