Imperial College London

How Does a Computer Perform Symbolic Mathematics?

Cecilia Della Casa, George Hilton, Weizhi Liu, Hanchong Zhu

June 2020

Contents

1 Introduction

1.1 Overview

Computer algebra systems allow the user to manipulate symbolic expressions with the fluidity of manual computation and the power of a computer. Tedious calculus operations, limits and evaluations become instant and correct. Some common examples include Wolfram Alpha[1], Mathematica[1] and Maple[2]; these are widely used by mathematicians and students alike. The named computer algebra systems have vast abilities from algebraically finding solutions to differential equations; infinite summation, product and expansion as well as plotting graphs in several dimensions.

We aim to develop our understanding of computer algebra systems by creating our own system in Python, able to perform differentiation, evaluation and simplification. The system should be able to take any function/expression (with variables over $\mathbb{R}$) and output the required result. In particular, we used object-oriented programming to implement said algorithms.

1.2 Data Structures

To implement our code we required the use of several data structures.

1.2.1 Directed Acyclic Graphs

The first, and arguably most fundamental, data structure we used belongs to the Graph Theory region of Mathematics - directed acyclic graphs[2]. Initially, we define a graph, a collection of vertices (nodes), with pairs of vertices connected by edges. We also define paths: a sequence of edges where the final vertex along an edge is the starting vertex for the next edge in the sequence. A cycle is a path which begins and ends at the same vertex. Thus, we can understand what a directed acyclic graph (abbreviated as DAG in all future references) is: a finite directed graph with no directed cycles. More simply, a graph is finite acyclic directed if there are finitely many vertices and edges and there is

[1] https://www.wolframalpha.com/

[2] Maple (2020). Maplesoft, a division of Waterloo Maple Inc., Waterloo, Ontario.

no non-trivial path from a vertex to itself.

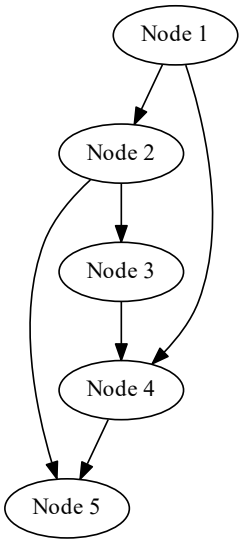


Figure 1.1: Here is a basic example of a DAG.

We are able to construct functions and derivatives using DAGs, see below for an example using $x^2 \sin(x^2)$.

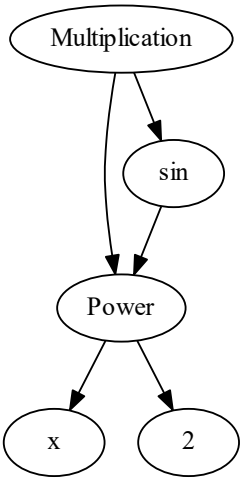


Figure 1.2: DAG of $x^2 \sin(x^2)$.

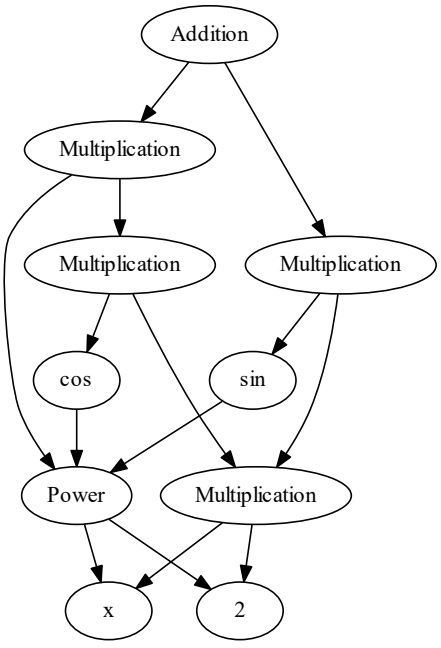


Figure 1.3: DAG of the derivative of $x^2 \sin(x^2)$

1.2.2 Trees and Their Comparison with DAGs

Another graph, which is widely used in computer science and mathematics, is trees[2]. A connected graph connects any two nodes by at least one path. Trees are acyclic connected graphs so, any two nodes in a tree are connected by a unique path. Developing this further, we define Binary Trees, a tree where each node has at most two 'children': a right and left child. The reason why we chose to use DAGs over trees will be clear when looking at the representation of $x^2 \sin(x^2)$.

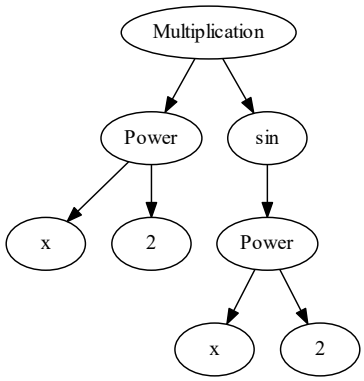


Figure 1.4: Binary tree of $x^2 \sin(x^2)$.

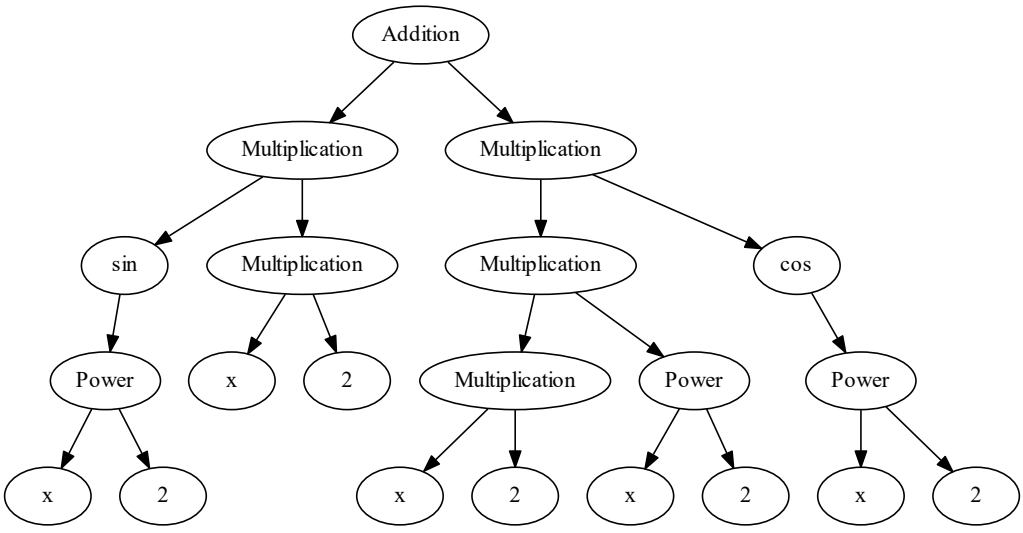


Figure 1.5: Binary tree of the derivative of $x^2 \sin(x^2)$

The downside of using trees is clear: in order to visit the top node in the graph it is necessary to visit all of the other nodes even though there are repetitions. To differentiate

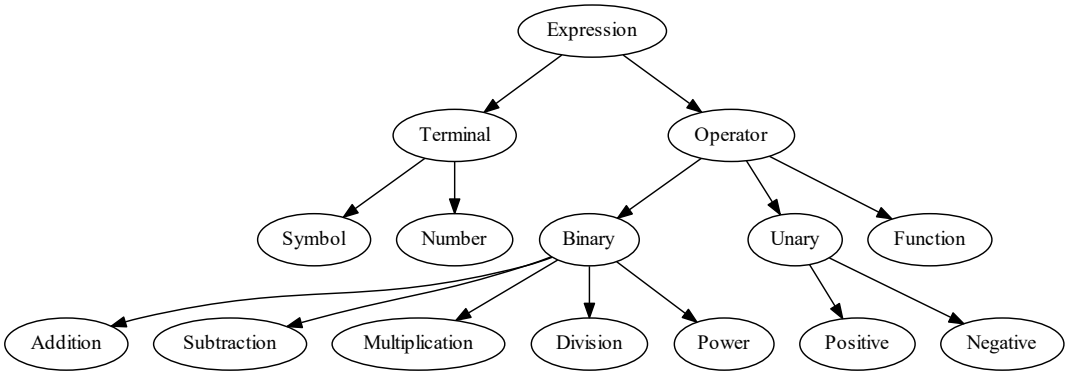
the above expression, for example, one would have to differentiate the left and right child of 'Addition' independently and so on. Hence, the differentiation tree will be larger and more complicated than the one resulting from differentiating the DAG. Implementing the DAG method is far more efficient - primarily by avoiding repetitions of the same operation and thus, our preferred method.

1.2.3 Objects and Classes

Let's consider the other data structures which we are going to use: objects and classes. Mathematical expressions will be modeled by objects, which are used to describe something that has specific characteristics and behaves in a certain way. In other words, 'an object is a collection of data with associated behaviours'[3]. It is possible to distinguish types of objects: x is not the same object as 2. It is also possible to distinguish objects of the same type, say, x and y.

Different types of objects are represented by different classes, which describe their attributes and behaviours. As before, x and y are different instances of the same class, so they have shared characteristics and behaviours but they are not equal to each other. Classes can also define methods: behaviours which can be performed on objects of that class. The official Python definition of classes reads as follows 'Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state'[4]. Classes are also a good model as we can create subclasses which inherit the characteristics and behaviours of their superclass; for example `Addition` is a type of `Binary` operation, so it will have the same behaviours and attributes as its superclass, `Binary`. This allows us to create a more compact and well-defined model for mathematical expressions as it is possible to define methods for a class that all of its subclasses will inherit.

We created a hierarchy of classes where each node of our DAGs would lie in a class, see below.



The most important relation between DAGs and our system of classes is the distinction between `Terminal`s and `Operator`s. `Terminal`s are the nodes that never have children, i.e. no edge originates from them; `Operator`s are the nodes from which edges start. In Figure 1.2 the only `Terminal`s are x and 2, all other nodes are `Operator`s or belong to one of its subclasses. This particular distinction will be fundamental for the implementation of our DAG visitor function: we will start visiting nodes from the bottom of the DAG, from the `Terminal`s, and build our way to the top: to visit `Operator`s.

2 Special Methods

2.1 Overview

The first thing we need to do is define the classes with their attributes and methods. Python can recognize and manipulate symbolic expressions through its special methods: these are predefined methods which define functions on our objects. Indeed, defining x as a `Symbol` creates an instance of the class `Symbol` through the special method `__init__(self, *operands)`; the expression $x + 1$ is recognized as the special method `__add__(self, other)` where x is self and 1 is other. These methods allow the user to input mathematical expressions that will be recognized and treated accordingly.

Every special method is surrounded by double underscores; we implemented several, as follows:

* `__init__(self, *operands)`: this is the special method used to initialise a class. In our model, every class is made up by operands, hence `__init__(self, *operands)` will establish these.
* `__add__(self, other)`,`__sub__(self, other)`, `__mul__(self, other)`, `__truediv__(self, other)`, `__pow__(self, other)`: these are the special methods that define the binary operations.
* `__pos__(self)`, `__neg__(self)`: these define the unary Operators + and -.
* `__repr__(self)`: this is the a special method used to return the string representation of self. Its purpose is to produce a more machine-readable output, in some cases even Python code[5].
* `__str__(self)`: this is the method that produces a human-readable output.

2.2 Special Methods for Representing Classes

We need to take a closer look at the two printing methods; they have different purposes as their outputs are supposed to be read by a different audience.

`__repr__(self)` outputs the serialisation of the graph of self (here self is the input); for

example the string representation of $(x+y)^x$ is `Pow(Add(Symbol('x'), Symbol('y')), Symbol('x'))`. We implemented this using a recursive algorithm, starting from the top of the DAG which returns its class name and the representation of its children in brackets[6].

Algorithm 1: __repr__ method

Output: representation of the object

1 **Function** `__repr__(`*self*`):`
2 $a \leftarrow$ class name
3 $b \leftarrow self.operand[0]$
4 **for** *o in operand[1:]* **do**
5 $b \leftarrow b + ',' + \text{repr}(o)$
6 **end**
7 **Return** $repr(a)$ + '(' + $repr(b)$ + ')'
8 **End Function**

Only one definition of this method is necessary so it will be under the Expression class and inherited by every other class.

Consider the previous example $(x+y)^x$: its DAG is in Figure 2.1; the first loop will return the string `Pow(repr(Add(x, y)), repr(x))`, the following loop returns `Pow(Add(repr(x), repr(y)), Symbol('x'))` and the final loop returns the desired string. This is why we say that `__repr__(self)` serialises the DAG.

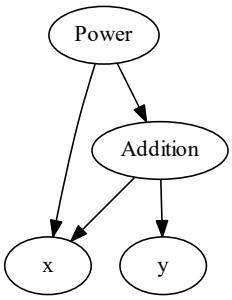


Figure 2.1: DAG of $(x+y)^x$

`__str__(self)` on the other hand should output the expressions as mathematicians would write them; this means being able to output a '+' whenever the class `Add` is invoked and so on. Therefore, it is necessary to define a symbol for the relevant classes and also, set up a way of dealing with the precedence of operations. For consistency we defined our operator precedence in the same way as Python's: see Table 2.1 below for the full list of symbols and operator precedence.

We need different definitions of `__str__(self)` based on how many operands the class takes. For `Terminal`s the definition is straightforward as it takes self as its input, and returns a string consisting of its operand. Under the other classes we have definitions that take care of the precedence issue: we defined it such that if the priority of the following operand is less than that of `self.operand` we add brackets[6]. For example if we want to print $x^{(y+z)}$, where x, y, z are predefined symbols, the output should be `x ** (y + z)` and not `x ** y + z` as `Add`'s priority is less than `Pow`'s.

Class	Priority	Symbol
Function(Operator)	6	function name (eg. 'sin')
Symbol(Terminal) Number(Terminal)	5	self.operand
Pow(Binary)	4	**
UAdd(Unary)	3	+
USub(Unary)		-
Mul(Binary)	2	*
Div(Binary)		/
Add(Binary)	1	+
Sub(Binary)		-

Table 2.1: priority table

Based on Table 2.1 the __str__ method is implemented as follows:

Algorithm 2: __str__ method for binary operators

Output: representation of the object

```
1  Function __str__(self):
2      if self.operand[0].priority <self.priority then
3          self.operand[0] ← '('+ str(self.operand)[0] +')'
4      else
5          self.operand[0] ← str(self.operand)[0]
6      end
7      if self.operand[1].priority <self.priority then
8          self.operand[1] ← '(' + str(self.operand)[1] +')'
9      else
10         self.operand[1] ← str(self.operand)[1]
11     end
12     Return self.operand[0] + self.symbol + self.operand[1]
13 End Function
```

Algorithm 3: __str__ method for unary operators and functions

Output: representation of the object

```
1  Function __str__(self):
2      if self.operand[0].priority <self.priority then
3          self.operand[0] ← '(' + str(self.operand)[0] + ')'
4      else
5          self.operand[0] ← str(self.operand)[0]
6      end
7      Return self.symbol + self.operand[0]
8  End Function
```

3 DAG Visitor

DAGs are used to maximise the efficiency of our symbolic algebra system. Many mathematical computations, such as differentiation and simplification are required to visit all nodes of the graph. Thus, the problem of avoiding the repetition of nodes arises. Unlike other symbolic representations, such as binary trees, where each node is only used once, DAG uses some nodes multiple times. Even though the general direction is from top to bottom, there may exist some nodes where more than one edge connects to it. Consider the previous example, there are two paths through the node 'x', namely Power $\longrightarrow$ x and Power $\longrightarrow$ Addition $\longrightarrow$ x.

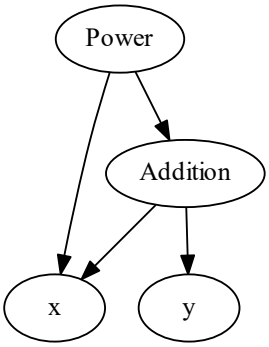


Figure 3.1: DAG of $(x + y)^x$, two different paths from the top power node to x node

To implement our DAG visitor function we use a `stack`: this is a linear data structure which stores items last in first out (LIFO) fashion[7]. When an item is appended to a `list`, the new item will be added to the last position. Furthermore, the only accessible item in the list is the last one added to it- insertions and deletions are known as `push(*)` and `pop()` respectively. We developed a DAG visitor function called `post_visit` which can be implemented using a `stack`. This takes as its arguments `self` and a more general `visitor_fn`[6]. Hence, this function, which incorporates a `stack`, can now be utilised for any operator thus, simplifying our code. Instead of defining new functions for derivative, evaluation and simplification we simply define methods under each class to implement `post_visit`.

Algorithm 4: Implementation of DAG visitor method

Input: An *expression* and a *visit_function*

Output: *visit_function*(*expression*)

```
1  Function post_visit(expression, visit_function):
2      visited ← {}
3      stack ← [expression]
4      while stack is not empty do
5          temp ← stackpop
6          to_visit ← [ ]
7          voperand ← [ ]
8          for n ← 0 to length(temp.operand) do
9              o ← temp.operand[n]
10             if o is a key of visited then
11                 voperand append visited[o]
12             else
13                 to_visit append o
14             end
15         end
16         if to_visit is not empty then
17             stack append temp
18             stack ← stack + to_visit
19         else
20             visited[o] ← visit_function(voperand)
21         end
22     end
23     Return visited[expression]
24 End Function
```

4 Automatic Differentiation

Automatic differentiation[8] is a useful technique in symbolic mathematics. We can differentiate a complex expression in an incredibly short time; implementation of this technique relies on DAGs. Each `Operator` (node) is equivalent to a mathematical function, i.e. an addition node $x + y$ can be consider as a function f such that $f = x + y$. Then the chain rule can be applied to DAGs in the same manner as it is applied to functions.

4.1 Chain Rule in the DAG Visitor Function

The chain rule is used when two functions are included in an expression in the following way: let, D_1, D_2, D_3 be subsets of $\mathbb{R}$ and let $g\colon D_1 \longrightarrow D_2$, $f\colon D_2 \longrightarrow D_3$. Defining $e(x) \coloneqq f(g(x))$ we have, $e'(x) = g'(x)f'(g(x))$.

In symbolic systems we consider functions, such as f and g, as different nodes i.e. to compute the derivative of a node, it is necessary to compute the derivative of its operands (nodes). However, by the chain rule, the derivative of the operands' operand is required. This process will not stop until there exists an `Operator` with `Terminal`-only operands. As seen in Figure 1.2, to obtain the derivative of the uppermost node `Mul`, the derivative of the `Pow` node and the `Sin` node are required. Meanwhile the derivative of the `Symbol` node and `Number` node are required for the `Pow` node. The chain stops here as `Symbol` and `Number` are the most basic nodes with a known derivative.

The most straightforward method to implement chain rule is a recursive algorithm. The derivative function of a node is defined by its own derivative multiplied by the derivative function of its operands. However, when the expression is complicated, the computation is very expensive as the time complexity is $O(2^n)$, where n is the nesting depth in shared expressions. To avoid this, we make use of the aforementioned DAG visitor function.

We define a dictionary to store the nodes and their derivatives. Meanwhile, the DAG visitor algorithm visits the nodes from bottom to top. If the system starts to differentiate the `Terminal`s, by applying the chain rule, the above nodes can be differentiated as well, since the derivative of its operands are known and stored in the `visited` dictionary. To execute this method we need to define the differentiation methods for each class.

4.2 Terminal Differentiation Methods

4.2.1 Symbol Method

During differentiation it is pivotal to consider which variable this operation is being performed with respect to. Different results will be obtained if the expression is differentiated with respect to different variables. In the `Symbol` only case, if x and y are two independent variables, then $\frac{dx}{dx} = 1$ and $\frac{dx}{dy} = 0$. So, for the `Symbol` differentiation method we have:

Algorithm 5: Differentiation of a symbol S with respect to x

Input: a symbol S which is going to be differentiated with respect to the symbol x

Output: the derivative of S

```
1 Function differentiation(S, x):
2     if S is x then
3         return 1
4     else
5         return 0
6     end
7 End Function
```

4.2.2 Number Method

For the derivative of `Number`, it is much more straightforward. The derivative of a constant - with respect to any variable - is always 0.

Algorithm 6: Differentiation of a number N with respect to x

Input: A number instance

Output: the derivative of this number

```
1 Function differentiation(N, x):
2     return 0
3 End Function
```

4.3 Operator Methods

The derivative of `Symbol` instances are the most basic part in a symbolic system. On the other hand, `Operator` is more complicated as `Operator` instances are built on `Terminal` instances. The main challenge with `Operator` differentiation is that each instance of `Operator` has its own rule for differentiation.

Although differentiation rules vary for different operators, the differentiation methods of operators have one common property: they take the derivative of their operands as an input. For example, $(\sin(x))' = x' \cos(x)$ and $(x + x)' = x' + x'$. The structure of the DAG visitor function solves this problem as it applies the chain rule from the bottom nodes upwards. Thus, the differentiation methods can be defined more easily.

Here is an example of how differentiation method is defined on the `Add` class:

Algorithm 7: Differentiation of `Add`

Input: *voperand* is a list defined in `post_visit`

var is the variable which user differentiate with respect to

Output: the derivative of Add operator

```
1 Function differentiation(self, voperand, var):
2 |   Return voperand[0] + voperand[1]
3 End Function
```

Table 4.1, on the next page, shows how differentiation of the `Operator` subclasses are defined.

Class	Operator	Differentiation
Unary	$+a$	$+\frac{da}{dx}$
	$-a$	$-\frac{da}{dx}$
Binary	$a+b$	$\frac{da}{dx}+\frac{db}{dx}$
	$a-b$	$\frac{da}{dx}-\frac{db}{dx}$
	a/b	$\frac{da}{dx}-\frac{db}{dx}$
	$a*b$	$\frac{da}{dx}-\frac{db}{dx}$
	$a**b$	$(\frac{db}{dx}\log a+\frac{da}{dx}\frac{b}{a})*a**b$
Functions	$\sin a$	$\frac{da}{dx}\cos a$
	$\cos a$	$-\frac{da}{dx}\sin a$
	$\exp a$	$\frac{da}{dx}\exp a$
	$\log a$	$\frac{da}{dx}\frac{1}{a}$

Table 4.1: Differentiation functions

4.4 Derivative

As we have defined the differentiation methods for each class we can now define our `derivative` function. It will take the place of `visitor_fn` in `post_visit`, making use of the DAG visitor method to maximise efficiency. Note, this is a higher-order function as it returns a function too.

Algorithm 8: Derivative

Input: an *expression* which is going to be differentiated with respect to *var*

Output: the derivative of *expression*

```
1 Function derivative(expression, var):
2     Function differentiation(voperand):
3         Return self.differentiation
4     End Function
5     Return post_visit(expression, differentiation)
6 End Function
```

5 Evaluation

Similar to what we did for differentiation, we aim to extend the use of the DAG visitor function for evaluation. In order to do so, we need to develop a function, namely `evaluation`, to once again take the place of `visitor_fn` in `post_visit`. This function evaluates expressions in an arbitrary number of variables. As for derivative, we need to define the evaluation methods for each class.

5.1 Number Method

Similar to differentiation, we start with the most basic class: `Number`. The evaluation of a number is just the value field defined with it. The result could be an `int` or `float` depending what `Number` instance we are evaluating.

5.2 Symbol Method

This is another basic subclass parallel to `Number`. Evaluating a symbolic expression depends on the desired value of the symbols in the expression; we must include said symbols and respective values in the function's input. A dictionary, namely `eva`, is a suitable data structure for storing such information.

To make it more user-friendly, we designed a set up function called `evalue`[6]. This function takes the symbols in the expression (as a `list`) and their values in the corresponding order (as a `list`), and outputs a dictionary with the same format as our `eva` definition. After initialising `eva`, the evaluation of `Symbol` instances should be straightforward: visit the dictionary `eva` with the keys being the symbols we want to evaluate. For consistency and ease of implementation later, we chose to output the value as an `int` or `float`. To demonstrate this see the following example:

```
In: evalue([x, y], [Number(1), Number(2)])
Out: {"Symbol('x')": Number(1), "Symbol('y')": Number(2)}
```

5.3 Operator Methods

Now we can progress to the evaluation of `Operator`. With the evaluated operands (Input value, stored in the list `voperand`) all being `int`/`float`, we applied the corresponding built-in operation onto those numbers and output `int`/`float` in return. For functions like `Sin`, `Cos` and `Log`, we chose to import the functions sin, cos and log in the `math` module and output a `float` as the result. `Operator` instances evaluation's are defined as follows:

Class	Operator	Evaluation
Unary	$+a$	$+a$
	$-a$	$-a$
Binary	$a + b$	$a + b$
	$a - b$	$a - b$
	a/b	a/b
	$a * b$	$a * b$
	$a * * b$	$a * * b$
Function	$\sin a$	$math. \sin a$
	$\cos a$	$math. \cos a$
	$\log a$	$math. \log a$

Table 5.1: Evaluation on Operator[6]

5.4 Evaluate

With the evaluation methods defined for the above classes, we compiled the code and created a function called `evaluate` which takes in the `Expression` we want to evaluate, the dictionary `eva`, and returns an `int` or `float`. More specifically, with the help of the prior established DAG visitor function, we don't evaluate the 'parents' before evaluating their children - we eventually obtain the evaluation of the uppermost node and return the result.

Algorithm 9: Evaluate

Input: an *expression* and a dictionary *eva*

Output: the evaluated *expression*

```
1 Function evaluate(expression, eva):
2     Function evaluation(voperand):
3         Return self.evaluation
4     End Function
5     Return post_visit(expression, evaluation)
6 End Function
```

6 Simplification

When testing the functionality of `derivative`, we noticed that it sometimes returned a rather disorganised result (but still mathematically correct). The following demonstrates a good example of this case:

```
print(derivative(x**Number(2), x))
2*x**(2 - 1)*1
```

Ideally we want `derivative(x**Number(2), x)` to output `2*x`. Inspired by this, we drafted a function which performs basic simplification on `Expression`, namely `simplify`[6].

6.1 Implementation

We realised we could again use the idea of DAG visitor; it would be efficient and clear to implement if we simplify the expression part by part. In order to do so, we need to define another function, called `simple`, to take the position of `visitor_fn` in `post_visit`. This function enacts the simplification process. As before, the simplification methods need to be defined in each class.

6.2 Terminal and Function Methods

For `Terminal` and `Function` methods, there is nothing to simplify, so we return pass on the original value.

6.3 Unary Method

The only instance we simplify is -0, which becomes 0.

6.4 Binary Operator Methods

The idea is to check if the input, `voperand`, meets certain requirement, and simplify accordingly.

Table 6.1 below shows how the `simple` function of each Binary operator is defined.

Class	Conditions	Simple
Add $a + b$	$a = 0$	b
	$b = 0$	a
	a, b both Number	Number$(a + b)$
	else	$a + b$
Sub $a - b$	$a = 0$	$-b$
	$b = 0$	a
	a, b both Number	Number$(a - b)$
	else	$a - b$
Mul $a * b$	$a = 1$	b
	$b = 1$	a
	$a = -1$	$-b$
	$b = -1$	$-a$
	$a = 0$ or $b = 0$	0
	a, b both Number	Number$(a * b)$
	else	$a * b$
Div a/b	$a = 0$	0
	$b = 1$	a
	$b = -1$	$-a$
	$b = 0$	*exception*
	a, b both Number	Number(a/b)
	else	a/b
Pow $a * * b$	$a = 0$	0
	$b = 1$	a
	$a = 1$ or $b = 0$	1
	a, b both Number	Number$(a * * b)$
	else	$a * * b$

Table 6.1: Simplification on Binary[6]

6.5 Simplify

As the `simple` methods are defined in each class, the `simplify` function can be created.

Algorithm 10: Simplify

Input: an *expression*

Output: the simplification of *expression*

```
1 Function simplify(expression):
2     Function simple(voperand):
3         Return self.simple
4     End Function
5     Return post_visit(expression, simple)
6 End Function
```

7 Testing

7.1 The Approach

It is critical that a symbolic system performs operations correctly or, as we expect them to. For the evaluation of a symbolic expression at a number, there is undoubtedly a correct answer: for example, $\sin(x)$ evaluated at $x = \pi$ gives 0. However, testing the accuracy of an expression relies on the system outputting a mathematically valid expression as expected by the user- this gives rise to a less formulaic approach.

We combined both methods to test our symbolic system. For `__repr__` we tested whether our actual output matched what we expected our system to give. For example, we would expect `(x**Number(2)).__repr__()` to output `Pow(Symbol('x'), Number(2))`.

Alternatively, for `__str__`, our `derivative` function and our `evaluate` function we were able to assess the validity of their outputs by comparison to the built in SymPy[9] module. SymPy is widely known and used- so we can take the outputs of its evaluation, differentiation and `__str__` method to be correct.

7.2 Implementation of the Testing

To complete the test we used the pytest module[3]. This test asserts that a boolean expression is true. To understand how this works, view the following example:

```
1 def func(x):
2     return x*2
3 def test_func():
4     assert func(10) == 20
```

In this example, our assertion is `assert func(10) == 20` - as this is `True`, pytest will report back a successful test. Our tests were more advanced and required predesignated fixtures and markings but the premise is the same as the simple case. The test can be run within our GitHub repository[6] or simply using Anaconda prompt.

[3]more information can be found here https://docs.pytest.org/en/5.4.3/index.html

7.3 Example of the Testing

To demonstrate the validity of our symbolic system we included one example in our test_.py file. Given the expression `x**2 + x*y` we created the following assertions:

1. `input_sym.__str__() == input_code.__str__()`

2. `"Add(Pow(Symbol('x'), Number(2)), Mul(Symbol('x'), Symbol('y')))" == input_code.__repr__()`

3. `str(s.simplify(s.derivative(input_code, x))) == str(s.diff(input_sym, sym.Symbol('x')))`

4. `s.evaluate(e, eva)` == `(sym.diff(input_sym, sym.Symbol('x'))).subs({sym.Symbol('x'): input_numberx, sym.Symbol('y'): input_numbery)`

for `__str__`, `__repr__`, `derivative` and `evaluate` respectively. For the full code and definitions see the repository[6]. This test is a good example of the versatility of our system: it achieves passes in each case yet, it involves multiple symbols and operators.

8 Summary

Symbolic algebra systems have revolutionised mathematics. Tedious computations are now instant and faultless; during our project we explored just one method yet there are many alternatives. We have developed our understanding of the theory and implementation of the DAG visitor function for performing symbolic differentiation - a powerful an efficient system. With more time it would be possible to continue expanding the system we created: developing our simplify function further - for example, so it recognises $x + x$ as $2x$ and $x.x$ as x^2.

Looking to the future it seems that the next step would be to incorporate integration into our system. However, symbolic integration is a great challenge and even the most advance systems across the world do not have a general method for definite integrals. The fundamental issue is that if we take a function, which can be represented in closed form, the same cannot necessarily be said about its antiderivative. A variation on the Risch Algorithm[10] can be used for indefinite integrals however, to solve definite integrals we need a variety of tricks such as, variable transformation, pattern matching and the incomplete gamma function[11].

The insight we gained into this vast region of computational mathematics was fascinating. The system is scalable and uses clever tricks to make the most daunting derivatives trivial. It would be interesting to develop our system further or even advance the techniques we gained. For example, automatic differentiation can be generalised to higher dimensions for gradients, hessians and jacobians[12]. Similarly, it forms an important part in computational statistics and machine learning- one notable example is Hamiltonian Monte Carlo sampling[13].

Acknowledgements

We thank Dr David Ham for his continued guidance and advice during our project - it was invaluable to us.

We also thank Shenghao Wu for the supplementary materials he provided.

References

[1] Wolfram Research I. Mathematica, Version 12.1;. Champaign, IL, 2020. Available from: https://www.wolfram.com/mathematica.

[2] Balakrishnan R, Ranganathan K. A textbook of graph theory. Springer Science & Business Media; 2012.

[3] Phillips D. Python 3 object-oriented programming. Packt Publishing Ltd; 2015.

[4] PYTHON SOFTWARE FOUNDATION, 9. Classes; 13/06/2020. Available from: https://docs.python.org/3/tutorial/classes.html#classes.

[5] Kettler R. A Guide to Python's Magic Methods. URL: https://rszalski github io/magicmethods/# intro. 2015;.

[6] Zhu H, Hilton G, Liu W, Della Casa C. M2R-AO3: How Does a Computer Perform Symbolic Mathematics? 2020 June;Available from: http://doi.org/10.5281/zenodo.3894612.

[7] Introduction to algorithms. Cambridge, Mass. ; London: MIT Press; 2009.

[8] Andersson J, Houska B, Diehl M. Towards a Computer Algebra System with Automatic Differentiation for use with Object-Oriented modelling languages. In: EOOLT; 2010. .

[9] Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, et al. SymPy: symbolic computing in Python. PeerJ Computer Science. 2017 Jan;3:e103. Available from: https://doi.org/10.7717/peerj-cs.103.

[10] Geddes KO, Czapor SR, Labahn G. In: The Risch Integration Algorithm. Boston, MA: Springer US; 1992. p. 511–573. Available from: https://doi.org/10.1007/978-0-585-33247-5_12.

[11] Moses J. Symbolic Integration: The Stormy Decade. Commun ACM. 1971 Aug;14(8):548–560. Available from: https://doi.org/10.1145/362637.362651.

[12] Kulisch U, Hammer R, Hocks M, Ratz D. In: Automatic Differentiation for Gradi-

ents, Jacobians, and Hessians. Berlin, Heidelberg: Springer Berlin Heidelberg; 1995. p. 244–292. Available from: https://doi.org/10.1007/978-3-642-79651-7_12.

[13] Hoffman MD, Gelman A. The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo; 2011.