Derivatives

Filip Lukowski

Spring Term 2023

Introduction

This report will look at finding derivatives of mathematical equations using the functional language of Elixir. To express a mathematical equation, an expression has to be implemented, which consists of numbers and variables, or what will be called literals. A literal can only be a number, which is a float or integer, or a variable as an atom, such as x or y. From literals, as well as using mathematical operators as atoms, a tuple for any equation can be made. For example, the line below represents $(2x + 3)^3$.

```
{:exp, {:add, {:mul, {:num, 2}, {:var, :x}}, {:num, 3}}, {:num, 3}}
```

Finding the Derivative

Now any mathematical equation can be expressed for the purpose of finding its derivative. There are various differentiation rules that need to be covered. Initially, the simple ones are done, such as derivative of a constant is 0, and derivative of a variable is 1 if the derivative is in respect to that variable, as seen below.

```
def deriv({:num, _}, _) do {:num, 0} end
def deriv({:var, v}, v) do {:num, 1} end
def deriv({:var, _}, _) do {:num, 0} end
```

Furthermore, the derivative of many terms added is the sum of their respective derivatives.

```
def deriv({:add, e1, e2}, v) do
    {:add, deriv(e1, v), deriv(e2, v)}
end
```

And lastly the product rule for multiplying expressions together.

```
def deriv({:mul, e1, e2}, v) do
    {:add,
      {:mul, deriv(e1, v), e2},
      {:mul, e1, deriv(e2, v)}
    }
  end
```

Slightly More Advanced

Now some slightly more advanced derivation techniques can be added. First there is the chain rule so equations wiht powers can be derived.

```
def deriv({:exp, e1, {:num, n}}, v) do
    {:mul,
      {:mul, {:num, n}, {:exp, e1, {:num, n-1}}},
      deriv(e1, v)
    }
end
```

Next is the quotient rule for dividing two expressions.

```
def deriv({:div, e1, e2}, v) do
    {:div,
      {:add,
        {:mul, deriv(e1, v), e2},
        {:mul, {:num, -1}, {:mul, deriv(e2, v), e1}}
      },
      {:exp, e2, {:num, 2}}
    }
end
```

Lastly, sin, cos and ln are also added.

```
def deriv({:ln, e1}, v) do
    {:div,
      deriv(e1, v),
      e1
    }
end
def deriv({:sin, e1}, v) do
    {:mul,
      deriv(e1, v),
      {:cos, e1}
    }
end
```

Not all trigonometric rules are included, such as tan or sec, however, these can be rearranged into expressions of sin and cos and more.

Simplifying

A derivative of any equation can now be found. Taking $(2x+3)^3$ as an example, the derivative returned looks the following way.

$$((3*(((2*x)+3))^2)*(((0*x)+(2*1))+0))$$

This is not very readable, even though it is correct. Thus, the equation returned should be simplified. This is perhaps the hardest task yet, as there are many rules, and a person can argue there will most likely be more ways, or more efficient ways. It was attempted that the oblivious terms were removed, such as ones multiplied by 0, or adding 0, as well as joining constants together wherever necessary. To simplify additions, the following functions are defined.

```
def simplify_add({:num, 0}, e2) do e2 end
def simplify_add(e1, {:num, 0}) do e1 end
def simplify_add{:num, n1}, {:num, n2} do {:num, n1 + n2} end
def simplify_add(e1, e2) do {:add, e1, e2} end
```

As seen above, there are 4 cases that can be simplified, such as for when adding to 0, the remaining value is the non 0 value. Adding two constant numbers returns their sum. Otherwise, both expressions are left as an add tuple.

Multiplication has many simplifications. The first few are straight forward, where it just returns 0 when multiplied by 0 or only one expression if multiplied by 1. Next come more complicated ones, such as product of the same variable returns the variable squared, or the multiplication of the same variable where one has a power component, as seen below.

```
def simplify_mul({:var, v}, {:exp, {:var, v}, {:num, n}}) do
    {:exp, {:var, v}, {:num, n+1}} end
```

Furthermore it had to be made sure that if an expression is a multiple of an expression and a number, and is to be multiplied by another number, then the returned value is the multiply tuple of the product of both numbers multiplied by the expression, as shown below.

```
def simplify_mul({:mul, e1, {:num, n1}}, {:num, n2}) do
    {:mul, {:num, n1*n2}, e1} end
```

There are many more cases for multiply, which mostly look similar to the one above just in different orders. Also, all the other possible results, including division, powers and ln, have quite straightforward simplifications involving work mainly with numbers of 0 and 1. Now the $(2x+3)^3$ equation returns a slightly nicer $(6*(((2*x)+3))^2)$.

The simplification is not perfect, as the brackets remain and can be confusing, as many of them do not have purpose. Also the program does not have simplified negatives, and it always shows $-1 * expression$, which is a limitation. Nevertheless, it still is a better version of the previous calculation.