ID1019 Assignment 1 Derivative

Yu Haihong

Spring Term 2023

Introduction

The goal of this assignment is to work with functions as symbolic expressions and compute the derivative of functions. The functions are represented as data structures in Elixir. Then, through a series of operations, the derivative is computed and output in a simplified form.

1 Literals and Expressions

The basic structures of functions are literals and expressions. Literals can be numbers, variables, or constants, which are defined by

```
@type literal() :: {:num, number()}
| {:var, atom()}
```

Meanwhile, expressions consist of arithmetic operations on literals. Considering only a few simple arithmetic operations, including addition, multiplication, exponentiation, logarithm, and trigonometric functions, expressions are defined as

```
@type expr() :: {:add, expr(), expr()}
| {:mul, expr(), expr()}
| {:exp, expr(), expr()}
| {:ln, expr(), expr()}
| {:sin, expr()}
| {:cos, expr()}
| literal()
```

2 Define Derivatives

The derivatives of expressions follow basic rules below. They cover basic operations on a single variable.

- $\frac{\mathrm{d}}{\mathrm{d}x}c = 0$

- $\frac{\mathrm{d}}{\mathrm{d}x}x = 1$
- $\frac{\mathrm{d}}{\mathrm{d}x}(f+g) = \frac{\mathrm{d}}{\mathrm{d}x}f + \frac{\mathrm{d}}{\mathrm{d}x}g$
- $\frac{\mathrm{d}}{\mathrm{d}x}(f*g) = \frac{\mathrm{d}}{\mathrm{d}x}f*g + f*\frac{\mathrm{d}}{\mathrm{d}x}g$
- $\frac{\mathrm{d}}{\mathrm{d}x}\mathrm{x}^{\mathrm{n}} = n\mathrm{x}^{\mathrm{n}-1}$
- $\frac{\mathrm{d}}{\mathrm{d}x}\ln x = \frac{1}{x}$
- $\frac{\mathrm{d}}{\mathrm{d}x}\sin x = \cos x,\ \frac{\mathrm{d}}{\mathrm{d}x}\cos x = -\sin x$
- $\frac{d}{dx}\left[f\left(u\right)\right] = \frac{d}{du}\left[f\left(u\right)\right]\frac{du}{dx}$

The function `deriv(expr, v)` is used to compute derivatives. Chain rule is applied for more complex combinations of expressions.

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

def deriv({:add, e1, e2}, v) do
  {:add, deriv(e1, v), deriv(e2, v)}
end
def deriv({:mul, e1, e2}, v) do
  {:add, {:mul, e1, deriv(e2, v)}, {:mul, e2, deriv(e1, v)}}
end
def deriv({:exp, e, {:num, n}}, v) do
  {:mul, deriv(e, v), {:mul, {:num, n}, {:exp, e, {:num, n-1}}}}
end
def deriv({:ln, e}, v) do
  {:mul, deriv(e, v), {:exp, e, {:num, -1}}}
end
def deriv({:sin, e}, v) do
  {:mul, deriv(e, v), {:cos, e}}
end
def deriv({:cos, e}, v) do
  {:mul, {:num, -1}, {:mul, deriv(e, v), {:sin, e}}}
end
```

After defining the derivative following of all the rules above, I can even perform more operations:

- Subtraction: $a - b = a + (-b)$
- Square root: $\sqrt{a} = a^{0.5}$
- Division and fraction: $a/b = \frac{a}{b} = a * b^{-1}$

3 Printout

As all the expressions are defined in tuples, it can be difficult to read the output. Therefore, a printout function `pprint(expr)` is defined. For example, {`:add, :ln, :var, :x, :cos, :var, :x`} can be print out as `(lnx + cosx)`. Example code such as

```
def pprint({:num, n}) do "#{n}" end
def pprint({:var, v}) do "#{v}" end
def pprint({:exp, e, {:num, n}}) do "#{pprint(e)} ^ #{n}" end
def pprint({:ln, e}) do "ln#{pprint(e)}" end
def pprint({:cos, e}) do "cos#{pprint(e)}" end
def pprint({:sin, e}) do "sin#{pprint(e)}" end
```

In addition and multiplication, as I introduced negative numbers, there might be a "+" followed by a "-" or "-1 *". Therefore, I simplified them by using `String.replace` method.

```
def pprint({:add, e1, e2}) do
  result = "(#{pprint(e1)} + #{pprint(e2)})"
  String.replace(result, "+ -", "- ")
end
def pprint({:mul, e1, e2}) do
  result = "#{pprint(e1)} * #{pprint(e2)}"
  String.replace(result, "-1 * ", "-")
```

4 Simplification

`simplify(expr)` is defined to simplify the expressions. Some simplification considerations include

- $x + 0 = x,\ 0 + x = x,\ n_1 + n_2 = n_3$ $(number\ addition)$
- $x{*}1 = x,\ 1{*}x = x,\ x{*}0 = 0,\ 0{*}x = 0,\ n_1{*}n_2 = n_3 (number\ multiplication)$
- $n_1 * (n_2 * v) = (n_1 * n_2) * v = n_3 * v$
- $v^1 = v,\ v^0 = 1$

5 Demonstration of Code

I defined a `test()` function in the module, with a pre-defined expression `e`. The function then calls `deriv()`, `simplify()`, and `pprint()`. Output examples are as follows.

```
iex(58)> Deriv.test()
original expression:
{:add, {:add, {:ln, {:var, :x}}, {:cos, {:var, :x}}},
 {:add, {:mul, {:num, 2}, {:exp, {:var, :x}, {:num, -2}}},
  {:add, {:mul, {:num, -3}, {:var, :x}}, {:num, 5}}}}
expression: ((lnx + cosx) + (2 * x ^ -2 + (-3 * x + 5)))
derivative: ((x ^ -1 - sinx) + (-4 * x ^ -3 - 3))
:ok
iex(60)> Deriv.test()
original expression:
{:exp, {:add, {:mul, {:num, -2}, {:var, :x}}, {:num, 1}}, {:num, 3}}
expression: (-2 * x + 1) ^ 3
derivative: -6 * (-2 * x + 1) ^ 2
```

6 Limitations

Due to insufficient time and the limit of my proficiency in Elixir programming, there are still many problems in this module.

First, the simplification regarding clashes of symbols is not solved sometimes, especially when they are separated by brackets. Usually, this is because the second symbol is attached to the latter part of the expression and is under multiple layers. The first demonstration showed this error.

Second, there isn't a very efficient way to input expressions. So far, the expressions are hard-coded inside `Deriv.test()`. It would be easier if users can input their expressions from console. However, this need to recognize a string as and input and convert it into a tuple form. Unfortunately, I don't have enough time to implement this function.

7 Links

The link to my code is in https://github.com/itshaihong/ID1019