# Derivatives

Ruxandra-Stefania Tudose

Spring Term 2024

## Introduction

This report aims to open up a window onto the very first assignment in functional programming, using Elixir, which consists in creating a program that given a function (within the limitations mentioned in the assignment) as input, computes its derivative.

## Approach: The Taken steps

### Step 1: Defining the literals and the expressions

The first step in tackling this task was understanding how the mathematical expressions should be represented in Elixir using atoms and tuples. By using these features and specifically atoms (which I perceive and understand as 'labels'), we can quickly identify what the given variable represents by grouping it in a tuple and together with pattern matching the next command that should be executed is therefore, identified and carried out. In order to be able to find the derivative of expressions such as **ln(f)** or **1/sin(f)**, new expressions have been defined as follows:

```
@type expr() :: {:ln, expr()}
| {:sqrt, expr()}
| {:sin, expr()}
| {:cos, expr()}
| {:div, expr(), expr()}
| {:div, literal(), expr()} [...]
```

### Step 2: Taking the derivative

As far as Calculus is concerned, there are certain rules that have to be obeyed when it comes to finding the derivative of a certain expression. Having said that, after having used pen and paper, I have 'translated' these rules using tuples in Elixir. When it comes to the derivative where a fraction is involved, I initially tried to avoid having to define a new operation, namely **:div** and

used negative powers. However, in the end, I came to the conclusion that it was quite difficult to read out (even simplified) the result. For each case individually, a 'function' has been defined.

Here is an example of how I defined one of the derivative, namely for the natural logarithm. In order to be able to take the derivative of a composed function as well, the whole expression has been multiplied with the recursive step of taking the derivative of the argument itself with respect to the variable we are differentiating over:

```
def deriv({:ln, e},v) do
  {:div, deriv(e,v), e}
end
#the derivative alternative I, eventually, gave up:
#{:mul, {:exp, e, {:num, -1}}, deriv(e,v)}
```

After having taken the derivative, the things that seemed a bit tricky were first its representation under the form of a tuple (which can be a difficult to read quickly) and second, the fact that it was not its simplified version (addition with zero, multiplication by one etc.)... which brings us to the next steps.

### Step 3: Simplifying the expressions | turning it into a string

Tuples are undoubtedly helpful, however they also imply quite an extensive number of parenthesis. In order to be able to read the result faster, we will turn it into a string using once again pattern matching. For each case individually, a 'function' has been defined. Here's an example of how I turned a division, into string a:

```
def pprint({:div, e1, e2}) do
   "(#{pprint(e1)})/(#{pprint(e2)})"
end
```

It should be taken into account that recursion and pattern matching is used for further writing the numerator and denominator under a string form as well.

### Step 4: Simplifying the expressions | the maths behind it

By using pattern recognition, several basic mathematical simplifications have been identified. Since there are cases in which the order doesn't matter *(e.g:* $0 + x = x + 0 = x$*)* and taking into consideration that there has to be a perfect match, all cases had to be taken into consideration. Moreover, more simplification 'functions' have been written for the situations when

the arguments of trigonometric or logarithmic functions have to be further calculated. Instead of writing new lines of code from scratch, I made use of the previously written ones, by simply adding a function that only accesses the arguments and then, calls the recursive step. One such example for the sine function is as follows:

```
def simplify({:sin, e1}) do
  {:sin, simplify(e1)}
end
```

Therefore, once I want to find the derivative of, for instance $sin(3x^2 + x + 13)$ the argument of the trigonometric function will be simplified as well. In other words, I will get rid of for instance, $0 * x^2$) etc. Therefore, the final result will look the following (taken from the terminal):

$$(cos((((3 * (x^2)) + x) + 13)) * ((6 * x) + 1))$$

## More on simplification and its current limitations

So far, I think that pattern matching is quite great since the worry related to the variable's type is no longer something of concern. On the other hand, it does impose a strict approach and model that has to be followed. Having said that, from my understanding so far, I think that pattern matching is the reason why in the end we cannot offer the simplest possible simplification of an obtained derivative. And that is because we would have to think of all the possible forms under which an expression could come. One such example I gave a try is $(sin(3x))^2$, whose 'simplest' result I got was:

$$((2 * sin((3 * x))) * (cos((3 * x)) * 3))$$

My assumption is that the reason I cannot multiply 2 and 3 in order to get one coefficient only, namely 6, is because they are part of different tuples. I would, therefore, have to correctly identify the numbers, extract them, multiply them and ultimately create a brand new tuple with the multiplication and the rest of the expressions. All in all, every time, new patterns have to be created and that is why I think there is no general solution to simplification. The tuple form (taken from the terminal):

```
{:mul,
   {:mul, {:num, 2}, {:sin, {:mul, {:num, 3}, {:var, :x}}}},
   {:mul, {:cos, {:mul, {:num, 3}, {:var, :x}}}, {:num, 3}}
}
```

## Overall Difficulties

Since up until this week, I have only dealt with static programming, at first it was a bit challenging to get used to the functional programming concept through Elixir. In other words, changing the paradigm involves changing the programming mindset. Pattern matching was also a bit tricky at first: variables no longer have a type and the only way to match them is by following the model.