CS 3430: Spring 2019: SciComp with Py
Assignment 1
Function Representation and Differentiation

Vladimir Kulyukin
Department of Computer Science
Utah State University

January 12, 2019

Learning Objectives

1. Function Representation
2. Function Differentiation
3. Differentiation Rules
4. Converting Function Representations to Python Functions
5. Graphing Functions and Derivatives

Introduction

In this assignment, you'll write Python functions that differentiate function representations, convert those representations to real Python functions, and graph them with matplotlib, one of the Python graphing libraries.

Warmup

If you're new to differentiation or your differentiation skills are rusty, I suggest that you start by reviewing the first two lectures and, if necessary, reading the handout, and then taking the derivatives of the functions below. If you're comfortable with differentiation, skip this section and move on.

1. $\frac{d}{dx}\left(6x^3\right)$; answer: $18x^2$;
2. $\frac{d}{dx}\left(3\sqrt[3]{x}\right)$; answer: $\frac{1}{x^{2/3}}$;

3. $\frac{d}{dx}\left(\frac{x}{2} - \frac{2}{x}\right)$; answer: $\frac{1}{2} + \frac{2}{x^2}$;
4. $\frac{d}{dx}\left(x^4 + x^3 + x\right)$; asnwer: $4x^3 + 3x^2 + 1$;
5. $\frac{d}{dx}(2x+4)^3$; answer: $6(2x+4)^2$;
6. $\frac{d}{dx}\left(3\sqrt[3]{2x^2+1}\right)$; answer: $\frac{4x}{(2x^2+1)^{2/3}}$;
7. $\frac{d}{dx}\left(\frac{1}{x^3+1}\right)$; answer: $\frac{-3x^2}{(x^3+1)^2}$;
8. $\frac{d}{dx}\left(\frac{2}{1-5x}\right)$; answer: $\frac{10}{(1-5x)^2}$;
9. $\frac{d}{dx}\left(2x + (x+2)^3\right)$; answer: $2 + 3(x+2)^2$;
10. $\frac{d}{dx}\left(\frac{4}{x^2}\right)$; answer: $\frac{-8}{x^3}$.

Variables, Powers, Sums, and Products

Recall that in lectures 1 and 2 we discussed how such mathematical objects as variables, powers, sums, and products can be represented with Python objects. The files `const.py`, `var.py`, `pwr.py`, `plus.py`, `prod.py` contain classes for constants, variables, powers, sums, and products, respectively. The file `maker.py` contains several functions for constructing objects of these classes. You don't need to modify these files for this assignment. Let's play with mathematical object construction a little in the Python interpreter window to get more comfortable with them.

Here's how we can construct a constant and get its value.

```
>>> c1 = make_const(1.0)
>>> c1.get_val()
1.0
>>> c2 = const(val=10.0)
>>> c2.get_val()
10.0
```

Once we have an object, we can use the function `isinstance` to check if the object is an instance of a given class. The two calls below show us that `c1` is an instance of the class `const` but not of the class `var` or the class `prod`.

```
>>> isinstance(c1, const)
True
>>> isinstance(c1, var)
False
>>> isinstance(c1, prod)
False
```

Here's how we can construct a list of constant objects and then extract their values into a new list.

```
>>> clist = [make_const(i) for i in range(10)]
>>> clist
>>> vlist = [c.get_val() for c in clist]
>>> vlist
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Let's represent the function x^2. This power expression has a base of x and a degree of 2. The base is an instance of the class `var` and the degree is an instance of the class `const`.

```
>>> fex = make_pwr('x', 2.0)
>>> b = fex.get_base()
>>> d = fex.get_deg()
>>> print b
x
>>> print d
2.0
>>> isinstance(b, var)
True
>>> isinstance(d, const)
True
```

Finally, let's represent $5x^2 + 10x - 100$.

```
>>> fex1 = plus(elt1=prod(mult1=make_const(5.0),
                          mult2=make_pwr('x', 2.0)),
                elt2=prod(mult1=make_const(10.0),
                          mult2=make_pwr('x', 1.0)))
>>> fex2 = plus(elt1=fex1, elt2=make_const(-100.0))
>>> fex2 = plus(elt1=fex1, elt2=make_const(-100.0))
>>> print fex2
(((5.0*(x^2.0))+(10.0*(x^1.0)))+-100.0)
```

Let's represent $5x^2$. The product's first multiple is 5 and its second multiple is x^2.

```
>>> fex = prod(mult1=make_const(5.0), mult2=make_pwr('x', 2.0))
>>> print fex
(5.0*(x^2.0))
>>> print fex.get_mult1()
5.0
>>> print fex.get_mult2()
(x^2.0)
```

Problem 1: (3 points)

Implement the function `deriv` that takes a function representation and computes a representation of its derivative. Your implemenation should be able to handle constants, powers, products, and sums. For this assignment, you should handle only simple products and simple powers. We'll handle more complex products, powers, and quotients later in the course.

A simple product can be recursively defined as a product of two constants, a constant and a simple power, a constant and a plus, and a constant and another simple product.

A simple power expression is recursively defined as an expression whose degree is a constant and whose base is a variable, a simple power, a sum, and a simple product. Below are a few test cases of how your implementation of `deriv` should work.

Let's start with x^1.

```
>>> fex = make_pwr('x', 1.0)
>>> print fex
(x^1.0)
>>> drv = deriv(fex)
>>> print drv
(1.0*(x^0.0))
```

Let's compute the derivative of another power.

```
>>> fex = make_pwr('z', 5)
>>> print fex
(z^5)
>>> drv = deriv(fex)
>>> print drv
(5*(z^4.0))
```

One more power expression, slightly more complex than the one above.

```
>>> fex = make_pwr_expr(make_pwr('x', 2.0), 2.0)
>>> print fex
((x^2.0)^2.0)
>>> drv = deriv(fex)
>>> print drv
((2.0*((x^2.0)^1.0))*(2.0*(x^1.0)))
```

Let's represent and differentiate $5x^2$ and x^25. Note that both expressions differentiate to the same expression.

```
>>> fex = prod(mult1=make_const(5.0), mult2=make_pwr('x', 2.0))
>>> drv = deriv(fex)
>>> print drv
(5.0*(2.0*(x^1.0)))
>>> fex = prod(mult1=make_pwr('x', 2.0), mult2=make_const(5.0))
>>> print fex
((x^2.0)*5.0)
>>> drv = deriv(fex)
>>> print drv
(5.0*(2.0*(x^1.0)))
```

Let's differentiate $(5x^{10})^4$ using the power rule.

```
>>> prd = prod(mult1=make_const(5.0),
               mult2=make_pwr('x', 10.0))
>>> fex = make_pwr_expr(prd, 4.0)
>>> print fex
((5.0*(x^10.0))^4.0)
>>> drv = deriv(fex)
>>> print drv
((4.0*((5.0*(x^10.0))^3.0))*(5.0*(10.0*(x^9.0))))
```

Here is another application of the power rule to differentiate $(x^3 + 3)^4$.

```
>>> fex = make_pwr_expr(plus(elt1=make_pwr('x', 3.0),
                             elt2=make_const(3.0)), 4.0)
>>> print fex
(((x^3.0)+3.0)^4.0)
>>> drv = deriv(fex)
>>> print drv
((4.0*(((x^3.0)+3.0)^3.0))*((3.0*(x^2.0))+0.0))
```

Let's differentiate a simple polynomial $x^2 + x - 100$.

```
>>> fex = plus(elt1=make_pwr('x', 2.0), elt2=make_pwr('x', 1.0))
>>> fex2 = plus(elt1=fex, elt2=make_const(-100.0))
>>> print fex2
(((x^2.0)+(x^1.0))+-100.0)
>>> drv = deriv(fex2)
>>> print drv
(((2.0*(x^1.0))+(1.0*(x^0.0)))+0.0)
```

There is some starter code for you in `deriv.py`. Write your code for Problem 1 in there.

Problem 2: (1 points)

Function representations are really useful but if we want to do scientific computing with them, we must be able to convert them into real Python functions. Write a function `tof` (this abbreviation stands for "to function") that takes a function expression (see Problem 1) and returns a Python function that actually computes the function represented by this expression. Here is a few test runs.

Let's work with $x^2 + x - 100$.

```
>>> fex = plus(elt1=make_pwr('x', 2.0), elt2=make_pwr('x', 1.0))
>>> fex2 = plus(elt1=fex, elt2=make_const(-100.0))
>>> print fex2
(((x^2.0)+(x^1.0))+-100.0)
```

Let's define a Python function that computes this polynomial.

```
>>> f = lambda x: x**2.0 + x - 100.0
>>> f
<function <lambda> at 0x7fcf2bf40050>
```

Let's run `tof` on `fex2` and save the function returned by `tof` in the variable `tf`. We also define a `test` function and test `f` and `tf` on $[0, 999]$.

```
>>> tf = tof(fex2)
>>> tf
<function f at 0x7fcf2bf400c8>
>>> def test():
        for i in range(1000):
            assert f(i) == tf(i)
print 'test passed'
>>> test()
test passed
```

Let's do another test with $5x^2$.

```
>>> fex = prod(mult1=make_const(5.0), mult2=make_pwr('x', 2.0))
>>> print fex
(5.0*(x^2.0))
>>> f = lambda x: 5.0*(x**2.0)
```

```
>>> tf = tof(fex)
>>> def test():
        for i in range(1000):
            assert f(i) == tf(i)
        print 'test passed'
>>> test()
test passed
```

There is some starter code for you in `tof.py`. Write your code for Problem 2 in there.

Problem 3: (1 points)

Let's finish it off by writing a function `graph_drv` that takes an function expression, differentiates it, converts both the function expression and the derivative expression into functions with `tof` and graphs both functions in the same plot. The function also takes a two-element list of floats, `xlim`, and another two element list of floats, `ylim`. These two lists specify the lower and upper bounds for the x-axis and y-axis, respectively, for the plot's grid. You should study the plotting code segments from the first two lectures (e.g., `lec01_01.py` and `lect02_03.py`) on how to use `xlim` and `ylim`.

Here is an example of applying `graph_drv` to the representation of $2x^5$. The plot of the function and its derivative for this test is shown in Fig. 1.

```
>>> prd = prod(mult1=make_const(2.0),
               mult2=make_pwr('x', 5.0))
>>> graph_drv(prd, [-3.0, 3.0], [-50.0, 50.0])
```

Let's apply `graph_drv` to the representation of $x^4 + x^3 + x$. The plot of the function and its derivative for this test is shown in Fig. 2.

```
>>> fex1 = make_pwr('x', 4.0)
>>> fex2 = make_pwr('x', 3.0)
>>> fex3 = make_pwr('x', 1.0)
>>> fex4 = plus(elt1=fex1, elt2=fex2)
>>> fex5 = plus(elt1=fex4, elt2=fex3)
>>> graph_drv(fex5, [-2.5, 2.5], [-10.0, 10.0])
```

There is some starter code for you in `graphdrv.py`. Write your code for Problem 3 in there.

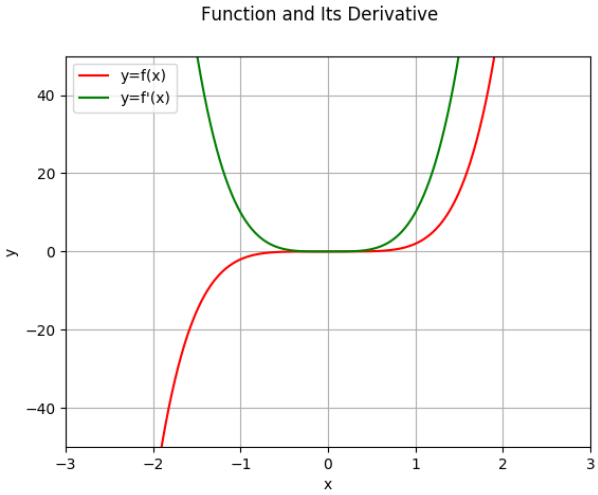

Figure 1: Plots of $2x^5$ and $10x^4$.

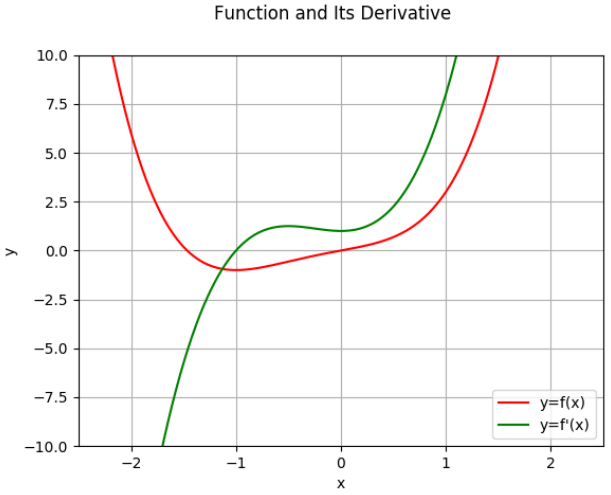

Figure 2: Plots of $x^4 + x^3 + x$ and $4x^3 + 3x^2 + 1$.

What to Submit

Submit the files `deriv.py`, `tof.py`, and `graphdrv.py` with your code via Canvas.

Happy Hacking!