CS 3430: SciComp with Py
Assignment 06
Newton-Raphson Algorithm

Vladimir Kulyukin
Department of Computer Science
Utah State University

February 15, 2020

Learning Objectives

1. Differentiation
2. Newton-Raphson Algorithm
3. Finding Zero Roots of Polynomials

Introduction

In this assignment, we'll implement a simple differentiation engine and use it to implement the Newton-Raphson algorithm to find zero roots of polynomials. We'll re-use `parser.py` and `tof.py` from the previous assignment to implement and test the engine. If you have trouble importing `parser` in your IDE, rename it to `my_parser` or `hw06_parser`. For your convenience, I've included in the zip all the auxiliary files (i.e., `maker.py`, `parser.py`, `prod.py`, `const.py`, `pwr.py`, `var`, and `plus.py`). I've tightened up the methods in `maker.py` with a few assertions on the basis of some common errors I saw in Assignment 05. These assertions are in place to make sure that you're passing the right types to the maker methods.

You'll code up your solutions to Problem 01 in `drv.py` and to Problem 02 in `nra.py`. Included in the zip is `cs3430_s20_hw06_uts.py` where I've written 40 unit tests you can test your code with as you implement it. Don't run all unit tests at once. Proceed one unit test at a time: comment them all out initially and then uncomment and run them one by one.

Problem 1: (1 point)

Let's implement the differention engine. The engine takes strings that encode functions (we'll confine ourselves to polynomials in this assignment), parses them into function representations, differentiates them, and returns the function representations of their derivatives. We'll use `tof.tof()` to convert the returned derivative representations to real Python functions.

Let's recall several simplifying assumptions from the previous assignment about string representations of polynomials. Each polynomial element is represented with the caret sign. For example, $5x^2$ is represented as `'5x^2'`, $10.5y^{-3}$ as `'10.5y^-3'`, $5.14z^3$ as `'5.14z^3'`. For the sake of parsing uniformity, constants are represented as products whose first multiplicand is the constant itself and whose second multiplicand is the polynomial's variable raized to the 0^{th} power. For example, 100 is represented as `'100x^0'`. Variables raised to the power of 1 are products of 1 and the variable raised to the power of 1. For example, x is represented as `'1x^1'`, y as `'1y^1'`, etc. Also, recall that there are no double minuses. To put it differently, we can't write polynomials as $5x^2 - -3x^3$. However, it's OK to have a coefficient's minus follow a plus: $5x^2 + -3x^3$. There are always spaces on both sides of '+' and '-' when they are between two elements of a polynomial.

The file `drv.py` contains the static method `drv.drv(expr)` that takes a function representation object `expr` (the one returned by `parser.parse_sum()`) or constructed manually with the `maker` methods and returns a function representation object of the derivative of the function represented by `expr`.

```
class drv(object):

    @staticmethod
    def drv(expr):
        if isinstance(expr, const):
            return drv.drv_const(expr)
        elif isinstance(expr, pwr):
            return drv.drv_pwr(expr)
        elif isinstance(expr, prod):
            return drv.drv_prod(expr)
        elif isinstance(expr, plus):
            return drv.drv_plus(expr)
        else:
            raise Exception('drv:' + str(expr))
```

As you can see from the above definition, our engine differentiates only constants, powers, products, and sums. Powers are `pwr` objects whose base is a `var` object and whose degree is a `const` object. It is also possible for the base to be a `pwr` object so long as its degree is a `const` object whose value is 1.0.

Implement `drv_const(expr)`, `drv_pwr(expr)`, `drv_prod(expr)`, and `drv_plus(expr)` that differentiate constants, powers, products, and sums, respectively. I left the assertions in place is the method stubs to remind you what data types you need to handle. By the way, each of the methods should be no more than 5 lines of code (a little longer if you use local variables to save intermediate results). So, if your definitions are longer, you're most likely doing something unnecessarily complicated. Make sure that the basics are taken care of and then let the loving wings of recursion carry you through. If you need to call one method of the `drv` class from another method, make sure that you prefix the name of the method with `drv.`, because these are static methods. For example, if you need to call `drv(expr)` from `drv_pwr(expr)`, do it as `drv.drv(expr)`.

Let's run some unit tests from `cs3430_s20_hw06_uts.py`. In the first unit test, we differentiate two constants, 1.0 and 153, to test `drv.drv_const(expr)`. Note that `drv.drv_const(expr)` returns a `const` object whose value is 0.

```
def test_hw06_prob01_ut01(self):
    print('\n***** CS3430: S20: HW06: Problem 01: Unit Test 01 ************')
    rslt = drv.drv_const(maker.make_const(1.0))
    err = 0.0001
    assert isinstance(rslt, const)
    assert abs(rslt.get_val() - 0.0) <= err
    rslt = drv.drv_const(maker.make_const(153.0))
    assert isinstance(rslt, const)
    assert abs(rslt.get_val() - 0.0) <= err
    print('CS 3430: S20: HW06: Problem 01: Unit Test 01: pass')
```

Unit tests 2, 3, and 4 test differentiation of variables. Remember that a variable is a product of 1 and the variable raized to the power of 1. Unit test 2 differentiates `'1x^1'` by parsing it with `parser.parse_sum()`, which returns a `prod` object whose first multiplicand is an `const` object (i.e., 1) and whose second multiplicand is a `pwr` object (i.e., x^1). The method `tof.tof()` is used to convert the function representation of the computed derivative to a Python function which is compared to the ground truth on a range of values.

```
def test_hw06_prob01_ut02(self):
    print('\n***** CS3430: S20: HW06: Problem 01: Unit Test 02 ************')
    s = '1x^1'
    fex = parser.parse_sum(s)
    print(fex)
    print(fex.get_mult2())
    print(drv.drv_pwr(fex.get_mult2()))
    gtf = lambda x: 1.0
    f = tof.tof(drv.drv_pwr(fex.get_mult2()))
    err = 0.0001
    for i in range(-100, 101):
        assert abs(gtf(i) - f(i)) <= err
    print('CS 3430: S20: HW06: Problem 01: Unit Test 02: pass')
```

Running unit test 2 produces the following output.

```
***** CS3430: S20: HW06: Problem 01: Unit Test 02 ************
(1.0*(x^1.0))
(x^1.0)
(1.0*(x^0.0))
CS 3430: S20: HW06: Problem 01: Unit Test 02: pass
```

Unit tests 5 – 10 test the differentiation of several polynomial elements each of which is a product of a constant and a power. For example, unit test 6 tests `drv.drv_prod()` to differentiate `'10x^4'`. We parse the string with `parser.parse_sum()`, create the ground truth function `gtf` that computes the derivative of $f(x) = 10x^4$ (i.e., $40x^3$), then compute the derivative of the function representation with `drv.drv_prod(fex)`, convert the derivative object to a Python function with `tof.tof()` and test the ground truth function and the function returned by `tof.tof()` on a range of values.

```
def test_hw06_prob01_ut06(self):
    print('\n***** CS3430: S20: HW06: Problem 01: Unit Test 06 ************')
    s = '10x^4'
    fex = parser.parse_sum(s)
    print(fex)
    print(drv.drv_prod(fex))
    gtf = lambda x: 40.0*x**3.0
    f = tof.tof(drv.drv_prod(fex))
    err = 0.0001
    for i in range(-100, 101):
        assert abs(gtf(i) - f(i)) <= err
    print('CS 3430: S20: HW06: Problem 01: Unit Test 06: pass')
```

Running unit test 6 produces the following output.

```
***** CS3430: S20: HW06: Problem 01: Unit Test 06 ************
(10.0*(x^4.0))
(10.0*(4.0*(x^3.0)))
CS 3430: S20: HW06: Problem 01: Unit Test 06: pass
```

The remainder of the unit tests (tests 11 – 20) test `drv.drv()` on various strings. For example, unit tests 20 differentiates `'5x^2 - 3x^5 + 10x^3 - 11x^4 + 1x^1 - 50x^0'`.

```
def test_hw06_prob01_ut20(self):
    print('\n***** CS3430: S20: HW06: Problem 01: Unit Test 20 ************')
    s = '5x^2 - 3x^5 + 10x^3 - 11x^4 + 1x^1 - 50x^0'
    fex = parser.parse_sum(s)
    print(fex)
    print(drv.drv(fex))
    gtf = lambda x: 10.0*x - 15.0*x**4.0 + 30.0*x**2 - 44.0*x**3 + 1.0
    f = tof.tof(drv.drv(fex))
    err = 0.0001
    for i in range(1, 21):
        assert abs(gtf(i) - f(i)) <= err
    print('CS 3430: S20: HW06: Problem 01: Unit Test 20: pass')
```

Running this unit test produces this output.

```
***** CS3430: S20: HW06: Problem 01: Unit Test 20 ************
((((((5.0*(x^2.0))+(-3.0*(x^5.0)))+(10.0*(x^3.0)))+(-11.0*(x^4.0)))
+(1.0*(x^1.0)))+(-50.0*(x^0.0)))
((((((5.0*(2.0*(x^1.0)))+(-3.0*(5.0*(x^4.0))))+(10.0*(3.0*(x^2.0))))
+(-11.0*(4.0*(x^3.0))))+(1.0*(1.0*(x^0.0))))+(-50.0*(0.0*(x^-1.0))))
CS 3430: S20: HW06: Problem 01: Unit Test 20: pass
```

Problem 2: (2 points)

The file `nra.py` contains the stubs of two static methods you'll implement to find zero roots of polynomials.

```
class nra(object):

    @staticmethod
    def zr1(fstr, x0, num_iters=3):
        pass

    @staticmethod
    def zr2(fstr, x0, delta=0.0001):
        pass
```

The method `zr1(fstr, x0, num_iters=3)` takes a string `fstr` with a polynomial, the first approximation to a zero root `x0`, and the number of iterations. It runs the Newton-Raphson algorithm for the specified number of iterations and returns the float zero root approximation found after the specified number of iterations.

In the method `zr2(fstr, x0, delta=0.0001)`, the first two arguments are the same as in `zr1(fstr, x0, num_iters=3)`. The third argument specifies the difference between two consecutive zero root values. This method keeps on computing zero root approximations until this difference is $\leq$ `delta`.

The file `nra.py` contains the following method for you to check how good a specific zero root value is.

```
def check_zr(fstr, zr, err=0.0001):
    return abs(tof.tof(parser.parse_sum(fstr))(zr) - 0.0) <= err
```

This method takes a string with a polynomial, `fstr`, and a zero root float value, `zr`, and returns true if the value returned by the Python function computed from the string and applied to `zr` is sufficiently close to 0.

The file `cs3430_s20_hw06_uts.py` contains 20 unit tests for Problem 2 to test finding zero roots of different polynomials. For example, unit tests 11 and 12 test `nra.zr1()` and `nra.zr2()`, respectively, to find a zero root of $300x^7 - 6x^4 - 30x^3 + 45x^2 + 7x + 10$.

```
def test_hw06_prob02_ut11(self):
    print('\n***** CS3430: S20: HW06: Problem 02: Unit Test 11 ************')
    s = '300x^7 - 6x^4 - 30x^3 + 45x^2 + 7x^1 + 10x^0'
    zr = nra.zr1(s, 5.0, num_iters=40)
    print('zr={}'.format(zr))
    assert nra.check_zr(s, zr, err=0.0001)
    print('CS 3430: S20: HW06: Problem 02: Unit Test 11: pass')

def test_hw06_prob02_ut12(self):
    print('\n***** CS3430: S20: HW06: Problem 02: Unit Test 12 ************')
    s = '300x^7 - 6x^4 - 30x^3 + 45x^2 + 7x^1 + 10x^0'
    zr, ni = nra.zr2(s, 5.0, delta=0.0001)
    print('zr={}; num_iters={}'.format(zr, ni))
    assert nra.check_zr(s, zr, err=0.0001)
    print('CS 3430: S20: HW06: Problem 02: Unit Test 12: pass')
```

Here's the output I got in Python 3.6.7 on Bionic Beaver (Ubuntu 18.04 LTS) with the inital zero root approximation of 5.0.

```
***** CS3430: S20: HW06: Problem 02: Unit Test 11 ************
zr=-0.752801290243841
CS 3430: S20: HW06: Problem 02: Unit Test 11: pass
.
***** CS3430: S20: HW06: Problem 02: Unit Test 12 ************
zr=-0.7528012902574428; num_iters=29
CS 3430: S20: HW06: Problem 02: Unit Test 12: pass
```

Of course, one can achieve faster conversions with better initial approximations which can be obtained by plotting functions.

What To Submit

Submit your code in `drv.py` and `nra.py`. It'll be easiest for us to grade your code if you place all the files (i.e., `var.py`, `const.py`, `pwr.py`, `plus.py`, `prod.py`, `pwr.py`, `maker.py`, `parser.py`, `tof.py`, `drv.py`, and `nra.py`) into one directory, zip it into `hw06.zip`, and upload your zip in Canvas.

Happy Hacking!