CS 3430: Scientific Computing
Assignment 06
Differentiation Engine and Newton-Raphson Algorithm

Vladimir Kulyukin
Department of Computer Science
Utah State University

February 19, 2022

Learning Objectives

1. Function Representation and Lambdification
2. Parsing Polynomial Strings into Function Representations
3. Differentiation and Differention Engines
4. Newton-Raphson Algorithm
5. Finding Zero Roots of Polynomials

Introduction

A fundamental theme of this assignment is function representation and lambdification. We'll play with function representations, figure out how to parse strings with polynomials into their function representations, and implement a simple lambdification algorithm to convert polynomial function representations into Python functions that we can actually execute.

You'll type and save your coding solutions for Problems 01 and 02 in `poly_parser.py` (polynomial parser) and `tof.py` (to function) included in the zip. Code for reuse, because we'll need these modules to implement a simple differentiation engine. This engine will be our exposure to symbolic mathematics in this class. Included in the zip is `cs3430_s22_hw06_uts.py` where I've written a few unit tests you can use to test your code with as you implement it.

Then we'll implement a simple differentiation engine and then use it in the Newton-Raphson algorithm (NRA) to find zero roots of polynomials. We'll use `poly_parser.py` and `tof.py` to implement and test the engine.

For your convenience, I've included in the zip all the auxiliary files (i.e., `maker.py`, `poly_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 the questions I was asked on Canvas or orally after class. These assertions are in place to make sure that you know which data types you're passing to the maker methods.

You'll code up your solutions to Problem 03 in `drv.py` and to Problem 06 in `nra.py`. Included in the zip is `cs3430_s22_hw06_uts.py` where I've written 40 unit tests (I think that's right, but I might have miscounted a couple) you can test your code with as you work on it. Proceed one unit

test at a time: comment them all out initially and then uncomment and run them one by one as you work on your solutions.

Let's take a deep dive into function representation that we'll need for Problems 01 and 02 of this assignment. Recall from Lecture 09 that a function representation is a data structure that represents a function. To put it differently, it contains all the necessary information about the function as a mathematical object that's needed to implement it. We need these representations for differentiation. The zip for this assignment contains the files `var.py`, `const.py`, `pwr.py`, `plus.py`, `prod.py`, `pwr.py`, and `maker.py` with the classes we discussed in Lecture 09.

Fire up Python on your computer, grab your favorite beverage, make yourself comfortable, and let's have some fun playing with these classes.

First off, we need to import them.

```
>>> from const import const
>>> from pwr import pwr
>>> from var import var
>>> from plus import plus
>>> from prod import prod
>>> from maker import maker
```

We can use the static methods of the maker class in `maker.py` to create a few variables. By the way, `isinstance` is a Python function that returns `True` if an object given in the first argument is an object of the class specified in the second argument.

```
>>> x = maker.make_var('x')
>>> y = maker.make_var('y')
>>> z = maker.make_var('z')
>>> isinstance(x, var)
True
>>> isinstance(y, var)
True
>>> isinstance(z, var)
True
```

Let's create a few constants.

```
>>> c1 = maker.make_const(val=1.0)
>>> c2 = maker.make_const(val=3.0)
>>> import math
>>> c3 = maker.make_const(val=math.sqrt(10))
>>> isinstance(c1, const)
True
>>> isinstance(c2, const)
True
>>> isinstance(c3, const)
True
>>> assert c1.get_val() == 1.0
>>> assert c2.get_val() == 3.0
>>> assert c3.get_val() == math.sqrt(10)
```

Let's make a few power expressions. Recall from Lecture 09 that a power expression is a variable raised to a real power exponent. In our representation, a power expression is an object whose base

is a variable (i.e., a `var` object) and whose exponent (i.e., degree) is a real constant (i.e., a `const` object). No complex numbers, because we don't want to go transcendental.

```
>>> p1 = maker.make_pwr('x', 2.0)
>>> p2 = maker.make_pwr('y', 2.0)
>>> p3 = maker.make_pwr('z', 5.0)
>>> isinstance(p1, pwr)
True
>>> isinstance(p2, pwr)
True
>>> isinstance(p3, pwr)
True
>>> isinstance(p1.get_base(), var)
True
>>> isinstance(p1.get_deg(), const)
True
>>> print(p1)
(x^2.0)
>>> print(p2)
(y^2.0)
>>> print(p3)
(z^5.0)
>>> p1.get_deg().get_val() == 2.0
True
>>> p2.get_deg().get_val() == 2.0
True
>>> p3.get_deg().get_val() == 5.0
True
```

A sum is a `plus` object that consists of two elements. The class that allows us to represent sums is in `plus.py`. By the way, we don't use the name `sum`, because it's the name of a Python function. Let's make us a few binary sum objects.

```
>>> sum1 = maker.make_plus(maker.make_const(1.0), maker.make_const(2.0))
>>> print(sum1)
(1.0+2.0)
>>> sum2 = maker.make_plus(maker.make_pwr('x', 2.0), maker.make_const(10.0))
>>> print(sum1)
(1.0+2.0)
>>> print(sum2)
((x^2.0)+10.0)
>>> isinstance(sum1.get_elt1(), const)
True
>>> isinstance(sum1.get_elt2(), const)
True
>>> isinstance(sum2.get_elt1(), pwr)
True
>>> isinstance(sum2.get_elt2(), const)
True
```

We can represent sums of arbirarily many elements as binary sums. For example, here's how we can represent $x^2 + x + 10$.

```
>>> sum2 = maker.make_plus(maker.make_plus(maker.make_pwr('x', 2.0),
                                           maker.make_pwr('x', 1.0)),
                           maker.make_const(10.0))
>>> print(sum2)
(((x^2.0)+(x^1.0))+10.0)
```

A product is an object that consists of two multiplicands. The class that allows us to represent products is defined in `prod.py`. Let's create $5x^2$ and $5x^2 - 10x^5$.

```
>>> p1 = maker.make_prod(maker.make_const(5), maker.make_pwr('x', 2.0))
>>> print(p1)
(5*(x^2.0))
>>> p2 = maker.make_prod(maker.make_prod(maker.make_const(5), maker.make_pwr('x', 2.0)),
                         maker.make_prod(maker.make_const(-10), maker.make_pwr('x', 5.0)))
>>> print(p2)
((5*(x^2.0))*(-10*(x^5.0)))
>>> print(p1.get_mult1())
5
>>> print(p2.get_mult2())
(-10*(x^5.0))
>>> print(p1)
(5*(x^2.0))
>>> print(p1.get_mult2())
(x^2.0)
>>> print(p2.get_mult1())
(5*(x^2.0))
>>> print(p2.get_mult2())
(-10*(x^5.0))
>>> print(p2.get_mult2().get_mult1())
-10
>>> print(p2.get_mult2().get_mult2())
(x^5.0)
>>> print(p2.get_mult2().get_mult2().get_deg())
5.0
```

We can use `plus` and `prod` objects to represent polynomials. Let's represent $5x^3 + 10x^2 + x + 100$.

```
elt1 = maker.make_prod(maker.make_const(5),
                       maker.make_pwr('x', 3))
elt2 = maker.make_prod(maker.make_const(10),
                       maker.make_pwr('x', 2))
elt3 = maker.make_prod(maker.make_const(1),
                       maker.make_pwr('x', 1))
elt4 = maker.make_const(100)
poly_sum = maker.make_plus(maker.make_plus(maker.make_plus(elt1, elt2),
                                           elt3),
                           elt4)
>>> print(poly_sum)
((((5*(x^3))+(10*(x^2)))+(1*(x^1)))+100)
>>> print(poly_sum.get_elt1())
(((5*(x^3))+(10*(x^2)))+(1*(x^1)))
>>> print(poly_sum.get_elt2())
100
```

```
>>> print(poly_sum.get_elt1().get_elt1())
((5*(x^3))+(10*(x^2)))
>>> print(poly_sum.get_elt1().get_elt1().get_elt2().get_mult2())
(x^2)
```

Problem 1 (1 point)

In this problem, we'll focus on function representation and lambdification. You'll type and save your coding solutions in `poly_parser.py` and `tof.py` included in the zip. Code for reuse, because we'll need these modules in the next assignment when we implement a simple differentiation engine. This engine will be our only exposure to symbolic mathematics in this class. Included in the zip is `cs3430_s22_hw06_uts.py` where I've written a few unit tests you can use to test your code with as you implement it.

Let's implement a simple parser that takes a string that represents a polynomial and converts it into a function representation. We won't implement a full blown symbolic math parser required for a real differentiation engine, which would be a semester long project, in and of itself, if we want to do it right. Rather, we'll confine ourselves to polynomials of the following form $a_n x^{r_n} + a_{n-1} x^{r_{n-1}} + ... + a_1 x^{r_1} + a_0 x^{r_o}$, where a_i and r_i are reals. Of course, we can use variables other than x in our polynomials. Thus, $5x^2$, $10.5y^{-3} + 5y^2 - 10y^{15} + 100$, $5.312z^{10} - 20.7z^{-2.5} + 4z^5 - 5.14z^3 + 10$ are valid polynomials.

We'll make a few simplifying assumptions about string representations of polynomials. We'll represent each polynomial element 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'`. To make our representation uniform, let's represent constants as products whose second multiplicand is the variable raized to the 0^{th} power. For example, 100 is represented as `'100x^0'`. Let's also represent variables raised to the power of 1 as products of 1 and the variable raised to the power of 1. Thus, x is represented as `'1x^1'`, y as `'1y^1'`, etc. Here are a few examples.

1. $5x^2$ is written as `'5x^2'`;
2. $10.5y^{-3} + 5y^2 - 10y^{15} + 100$ is written as `'10.5y^-3 + 5y^2 - 10y^15 + 100x^0'`;
3. $5.312z^{10} - 20.7z^{-2.5} + 4z^5 - 5.14z^3 + z + 10$ is written as `'5.312z^10 - 20.7z^-2.5 + 4z^5 - 5.14z^3 + 1z^1 + 10z^0'`.

Let's further agree not to have double minuses. In other words, we wont' have polynomials written as $5x^2 - -3x^3$. However, it's fine to have a coefficient with a minus follow a plus (e.g., $5x^2 + -3x^3$) although $5x^2 - 3x^3$ would be easier.

Implement the static method `poly_parser.parse_elt(elt)` in the `parser` class that takes a string representation of a polynomial element and converts it into a `prod` object. Here's a test run.

```
s1 = '5x^10'
e1 = poly_parser.parse_elt(s1)
```

If our implementation is correct, all the assertions below pass (see `test_hw06_prob01_ut01()` in `cs3430_s22_hw06_uts.py`).

```
err = 0.0001
assert str(e1) == '(5.0*(x^10.0))'
assert isinstance(e1, prod)
assert isinstance(e1.get_mult1(), const)
assert abs(e1.get_mult1().get_val() - 5.0) <= err
```

```
assert isinstance(e1.get_mult2(), pwr)
assert isinstance(e1.get_mult2().get_base(), var)
assert e1.get_mult2().get_base().get_name() == 'x'
assert isinstance(e1.get_mult2().get_deg(), const)
assert abs(e1.get_mult2().get_deg().get_val() - 10.0) <= err
```

Here's another test (see `test_hw06_prob01_ut04()` in `cs3430_s22_hw06_uts.py`).

```
s1 = '1001.7341z^-11.57'
e1 = poly_parser.parse_elt(s1)
```

All the assertions below pass.

```
err = 0.0001
assert isinstance(e1, prod)
assert isinstance(e1.get_mult1(), const)
assert abs(e1.get_mult1().get_val() - 1001.7341) <= err
assert isinstance(e1.get_mult2(), pwr)
assert isinstance(e1.get_mult2().get_base(), var)
assert e1.get_mult2().get_base().get_name() == 'z'
assert isinstance(e1.get_mult2().get_deg(), const)
assert abs(e1.get_mult2().get_deg().get_val() + 11.57) <= err
assert str(e1) == '(1001.7341*(z^-11.57))'
```

On to sums. Implement the static method `parse_sum(poly_str)` in `poly_parser.py` that takes a string representation of a polynomial written according to the above conventions and returns a `plus` object representing this polynomial.

Here's a unit test (see `test_hw06_prob01_ut05()` in `cs3430_s22_hw06_uts.py`).

```
s1  = '5x^-10 + 3x^5'
sum_ex  = poly_parser.parse_sum(s1)
```

All the assertions below pass.

```
err = 0.0001
assert isinstance(sum_ex, plus)
assert isinstance(sum_ex.get_elt1(), prod)
assert isinstance(sum_ex.get_elt2(), prod)
e1 = sum_ex.get_elt1()
assert abs(e1.get_mult1().get_val() - 5.0) <= err
assert isinstance(e1.get_mult2(), pwr)
assert isinstance(e1.get_mult2().get_base(), var)
assert e1.get_mult2().get_base().get_name() == 'x'
assert isinstance(e1.get_mult2().get_deg(), const)
assert abs(e1.get_mult2().get_deg().get_val() + 10.0) <= err
assert str(e1) == '(5.0*(x^-10.0))'
e2 = sum_ex.get_elt2()
assert abs(e2.get_mult1().get_val() - 3.0) <= err
assert isinstance(e2.get_mult2(), pwr)
assert isinstance(e2.get_mult2().get_base(), var)
assert e2.get_mult2().get_base().get_name() == 'x'
assert isinstance(e1.get_mult2().get_deg(), const)
assert abs(e2.get_mult2().get_deg().get_val() - 5.0) <= err
assert str(e2) == '(3.0*(x^5.0))'
```

More unit tests are in `cs3430_s22_hw06_uts.py`.

Problem 2 (1 point)

Our function representations are not that useful unless we can convert them into Python functions that we can run. Let's address this issue and implement the static method `tof.tof(ex)` in `tof.py` (`tof` stands for "to function") that takes a function representation `fr` returned by `poly_parser.parse_sum()` or `poly_parser.parse_elt()` and returns a Python function that corresponds to that representation. This method should handle `const`, `var`, `prod`, `pwr`, and `plus` objects. If `fr` is not any of these, this static method throws an exception. Here's a unit test (see `test_hw06_prob02_ut01()` in `cs3430_s22_hw06_uts.py`).

```
from tof import tof
from poly_parser import poly_parser
ex = '5x^2'
my_fun = tof.tof(poly_parser.parse_sum(ex))
```

Let's define a ground truth function (`gt_fun`) and compare its output with the output of the function returned by `tof.tof()`.

```
gt_fun = lambda x: 5.0*(x**2.0)
err = 0.0001
for x in range(-1000000, 1000000):
    assert abs(my_fun(x) - gt_fun(x)) <= err
```

Here's another unit test (see `test_hw06_prob02_ut03()` in `cs3430_s22_hw06_uts.py`).

```
ex = '5x^-2 - 3x^5 + 4.5x^7.342'
my_fun = tof.tof(poly_parser.parse_sum(ex))
```

Again, we define a ground truth function (`gt_fun`) and compare its output with the output of the function returned by `tof.tof()`.

```
gt_fun = lambda x: 5.0*(x**-2.0) - 3.0*(x**5.0) + 4.5*(x**7.342)
err = 0.0001
for x in range(-1000000, 0):
    assert abs(my_fun(x) - gt_fun(x)) <= err
for x in range(1, 1000000):
    assert abs(my_fun(x) - gt_fun(x)) <= err
```

Problem 3 (1 point)

Let's move on to the differention engine. The engine takes strings that encode functions (recall that we'll confine ourselves to polynomials), parses them into function representations, differentiates them, and returns the function representations of their derivatives. We'll use `tof.tof()` to convert the returned function representations of derivatives to real Python functions (i.e., to lambdify these function representations).

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 (and simplification!), we'll represent constants 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'`. Actually, since we're dealing with polynomials in one variable, the variable names are irrelevant insomuch as the polynomial $5x^2 + 7x + 10$ is as good as the polynomial $5z^2 + 7z + 10$. In fact, solutionwise, they are the same. Bottom line – choose your favorite variable name (e.g., `'x'`) and use it consistently in each polynomial. What I mean is that each polynomial must have the same variable in each of its elements. In other words, $5z^2+7z+10$ and $5x^2+7x+10$ are great, but $5x^2+7z+10$ is not, because its a polynomial in 2-variables, which we won't play with.

A variable raised to the power of 1 is represented as a product of 1 (a constant object) and a power object of the variable (a variable object) raised to the power of 1 (a constant object). In writing, this can be reflected by writing x `'1x^1'`, y as `'1y^1'`, etc.

Also, recall that we've agreed that there are no double minuses. Said another way, we won't write polynomials as $5x^3 - -3x^3$ ($5x^3 + 3x^2$ is more beautiful, in my humble opinion, anyway – not as much negativity). It's OK, however, to have a coefficient's minus follow a plus: $5x^2 + -3x^3$. Let's agree that there are always spaces on both sides of `'+'` and `'-'` when they are between two elements of a polynomial. Recall that if `s` is a string with a polynomial description, we can use `s.split()` to split `s` on white space to obtain string representations of individual elements.

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

```
class drv(object):

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

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. Remember that, depending on your implementation, 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. In other words, $(x^1)^2$ is mathematically identical to x^2. That's the representation that I use in my implementation, because I find that the application of the power rule to x^1 is easy to code in the sense that the derivative of x^1 can be represented as a product object whose first element is 1 (a constant object) and whose second element is a power object whose base is a variable object (i.e., x) and whose degree is a constant object (i.e., 0). I just realized that what I just wrote can be expressed much more succinctly as $(x)' = (x^1)' = 1 \cdot x^0$. But, let's move on.

Implement `drv_const(fr)`, `drv_pwr(fr)`, `drv_prod(fr)`, and `drv_plus(fr)` that differentiate constants, powers, products, and sums, respectively. I left the assertions in place in the method stubs to remind you what data types you need to handle.

By the way, let me note, in passing, that each of the methods should be no more than 5 lines of code (well, a little longer if we use local variables to save intermediate results). So, if you find yourself

writing longer definitions, you're most likely doing something unnecessarily complicated. Make sure that the basic cases are taken care of and then let the loving wings of recursion left you up and carry you over the deserts of uncertainty.

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(fr)` from `drv_pwr(expr)`, do it as `drv.drv(fr)`.

Let's run some unit tests from `cs3430_s22_hw06_uts.py`. In the first unit test, we differentiate two constants, 1.0 and 153, to test `drv.drv_const(fr)`. Note that `drv.drv_const(expr)` returns a `const` object whose value is 0.

```
def test_hw06_prob03_ut01(self):
    print('\n***** CS3430: S22: HW06: Problem 03: 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: S22: HW06: Problem 03: 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 lambdify the function representation of the computed derivative into a Python function which is compared to the ground truth on a range of values.

```
def test_hw06_prob03_ut02(self):
    print('\n***** CS3430: S22: HW06: Problem 03: Unit Test 02 ************')
    s = '1x^1'
    fr = poly_parser.parse_sum(s)
    print(fr)
    print(fr.get_mult2())
    print(drv.drv_pwr(fr.get_mult2()))
    gtf = lambda x: 1.0
    f = tof.tof(drv.drv_pwr(fr.get_mult2()))
    err = 0.0001
    for i in range(-100, 101):
        assert abs(gtf(i) - f(i)) <= err
    print('CS 3430: S22: HW06: Problem 03: Unit Test 02: pass')
```

Running unit test 2 produces the following output.

```
***** CS3430: S22: HW06: Problem 03: Unit Test 02 ************
(1.0*(x^1.0))
(x^1.0)
(1.0*(x^0.0))
CS 3430: S22: HW06: Problem 03: 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(fr)`, 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_prob03_ut06(self):
    print('\n***** CS3430: S22: HW06: Problem 03: Unit Test 06 ************')
    s = '10x^4'
    fr = poly_parser.parse_sum(s)
    print(fr)
    print(drv.drv_prod(fr))
    gtf = lambda x: 40.0*x**3.0
    f = tof.tof(drv.drv_prod(fr))
    err = 0.0001
    for i in range(-100, 101):
        assert abs(gtf(i) - f(i)) <= err
    print('CS 3430: S22: HW06: Problem 03: Unit Test 06: pass')
```

Running unit test 6 produces the following output.

```
***** CS3430: S22: HW06: Problem 03: Unit Test 06 ************
(10.0*(x^4.0))
(10.0*(4.0*(x^3.0)))
CS 3430: S22: HW06: Problem 03: 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_prob03_ut20(self):
    print('\n***** CS3430: S22: HW06: Problem 03: Unit Test 20 ************')
    s = '5x^2 - 3x^5 + 10x^3 - 11x^4 + 1x^1 - 50x^0'
    fr = poly_parser.parse_sum(s)
    print(fr)
    print(drv.drv(fr))
    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(fr))
    err = 0.0001
    for i in range(1, 21):
        assert abs(gtf(i) - f(i)) <= err
    print('CS 3430: S22: HW06: Problem 03: Unit Test 20: pass')
```

Running this unit test produces this output.

```
***** CS3430: S22: HW06: Problem 03: 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: S22: HW06: Problem 03: Unit Test 20: pass
```

Problem 4 (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 (NRA) 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_s22_hw06_uts.py` contains 20 unit tests for Problem 4 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_prob04_ut11(self):
    print('\n***** CS3430: S22: HW06: Problem 04: 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: S22: HW06: Problem 04: Unit Test 11: pass')

def test_hw06_prob04_ut12(self):
    print('\n***** CS3430: S22: HW06: Problem 04: 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: S22: HW06: Problem 04: Unit Test 12: pass')
```

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

```
***** CS3430: S22: HW06: Problem 04: Unit Test 11 ************
zr=-0.752801290243841
CS 3430: S22: HW06: Problem 04: Unit Test 11: pass
.
***** CS3430: S22: HW06: Problem 04: Unit Test 12 ************
zr=-0.7528012902574428; num_iters=29
CS 3430: S22: HW06: Problem 04: Unit Test 12: pass
.
```

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

The Famous $\sqrt{2}$

I'd like to draw your attention to `test_hw06_prob04_ut06()` in `cs3430_s22_hw06_uts.py` where the NRA is used to approximate $\sqrt{2}$ by finding a zero of $x^2 - 2$.

```
def test_hw06_prob04_ut06(self):
    print('\n***** CS3430: S22: HW06: Problem 04: Unit Test 06 ************')
    s = '1x^2 - 2x^0'
    zr, ni = nra.zr2(s, 1.0, delta=0.0001)
    print('zr={}; num_iters={}'.format(zr, ni))
    assert nra.check_zr(s, zr, err=0.0001)
    print('CS 3430: S22: HW06: Problem 04: Unit Test 05: pass')
```

There's a story here, both sad and inspiring. The ancient Greek mathematicians and astronomers (especially, the Pythagoreans) believed that the natural numbers (and they did not consider 0 as a natural number) were the only true numbers, because they were "God-given" [1]. They believed that the rational numbers were ratios of natural numbers and were gapless. In fact, the Pythagoreans viewed the rational numbers as a continuous "flow of quantity."

All was well until a member of the Pythagorean Brotherhood, long after the death of Pythagoras himself, decided to measure the length of the diagonal of a unit square. I should note that although the Pythagoreans called themselves the *Brotherhood*, women were completelly equal to men among the Pythagoreans. In fact, Pythagoras' wife, Theano, became the leader of the Brotherhood after Pythagoras' death and wrote several important papers on various aspects of mathematics.

The Pythagorean scholar who decided to measure the length of the diagonal in the unit sqquare knew the Pythagorean theorem, of course, and so he reckoned that if h is the length of the diagonal, then $h^2 = 1 + 1$ and $h = \sqrt{2}$. Since the rationals were considered gapless, he further reckoned that $h = \sqrt{2} = m/n$, for some natural numbers m and n (recall that 0 was not considered a natural number). Thus, since $\sqrt{2} = m/n$, one had to have $m^2 = 2n^2$. He also knew that Euclid, by that time, had already shown that every natural number greater than 1 has a unique prime factorization (i.e., can be represented as a unique product of prime numbers). Since the scholar was squaring two natural numbers m^2 and n^2, every prime factor of m and n had to appear in the prime factorizations of m^2 and n^2 a even number of times. But then the number of times that 2 appears in $2n^2$ is odd. B-bbb-o-o-o-ooo-ooo-m-mmm!!! The mental detonation in that scholar's mind must have been loud! If $m^2 = 2n^2$ and every number has a unique prime factorization, the number of times 2 occurs in

m^2 must be the same as the number of times it occurs in $2n^2$, but that, alas, was not the case. This famous argument is the foundation of the proof that $\sqrt{2}$ is not a rational number.

The Pythagoreans did not treat well the discovery that $\sqrt{2}$ was not rational. They called it $\alpha\lambda o\gamma o\sigma$ (alogos), which means "unspeakable" or "inexpressible." Some historians of mathematics believe that this was the reason why numbers such as $\sqrt{2}$ were later called *irrational* (unreasonable). Note a fine touch of neurolinguistic conditioning in this term: $\sqrt{2}$ was discovered by human reason (ratio) yet is called unreasonable (irrational).

The Pythagoreans took an oath to never share this knowledge with people outside of their Brotherhood. Anyone who would reveal this knowledge to the public would be put to death. A member the Brotherhood (we don't know if it was the same member who proved the irrationality of $\sqrt{2}$ or a different person) revealed this knowledge to the public. A Greek philosopher of the 5-th century B.C.E writes that the man who "fortuitously revealed this aspect of the living things" to the world perished in a shipwreck. Other historians of mathematics commenting on this episode write that that member of the Pythagorean Brotherhood was put to death by drowning.

This story is sad, because a mathematician paid with his or her life for speaking the truth. However, it is inspiring and, indeed, "fortuitous" to all of us, because centuries later the Greak astronomer and mathematician Eudoxus, a student of Plato, offered the first (currently known to historians of mathematics) definition of irrational numbers, thus foreshadowing the work of the great Georg Cantor in the second half of the 19-th century. And Cantor showed that irrational numbers are much more numerous and common than rational numbers (and, consequently, as one may say, more reasonable than the numbers we call reasonable).

When I run this unit test on my laptop, I get the following output.

```
***** CS3430: S22: HW06: Problem 04: Unit Test 06 ************
zr=1.4142135623746899; num_iters=3
CS 3430: S22: HW06: Problem 04: Unit Test 05: pass
```

I get the following value with `numpy`.

```
>>> import math
>>> math.sqrt(2)
1.4142135623730951
```

The two approximations of the famous $\sqrt{2}$ (i.e., 1.4142135623746899 and 1.4142135623730951) agree on the first 11 digits after the decimal.

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`, `poly_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!

References

[1] E. Burger. "Zero to Infinity: A History of Numbers." The Great Courses Publishing, 2007.