CS 3430: S24: Scientific Computing
Assignment 05
Differentiation Engine and Newton-Raphson Algorithm

Vladimir Kulyukin
Department of Computer Science
Utah State University

February 19, 2024

Learning Objectives

1. Differentiation and Differention Engines
2. Symbolic Mathematics
3. Newton-Raphson Algorithm
4. Finding Zero Roots of Polynomials

Introduction

A fundamental theme of this assignment is symbolic mathematics, and, in particular, differentiation and differentiation engines (DEs). We will implement a tiny DE that we discussed in the last three lectures. If you need to, you should scan through the PDFs of lectures 08, 10, and 11 and/or your class notes.

We will pay close attention to function representation and lambdification. In scientific computing, function representation typically refers to the programmatic description of functions with data structures, e.g., tuples, lists, arrays, trees, instances of classes, etc. Lambdification is the process of turning function representations into executable code segments. These generated code segments are usually, but not always, functions. I said *not always* in the previous sentence, because if we a turning, e.g., Python function representations into Java code segments, we will most likely be generating Java classes with methods that are functional equivalents of the Python data structures. So, we will play with function representations, figure out how to parse texts with polynomials into their function representations, and work with lambdification and differentiation methods to convert polynomial representations and derivatives thereof into Python functions that we can execute.

Then we will use our tiny DE to implement the Newton-Raphson algorithm (NRA) to find zero roots of polynomials. We will also play with `sympy`, a great Python library on which our tiny DE is modeled. The full mastery of `sympy` requires a separate course. The library is not limited just to differentiation. We can integrate, approximate, interpolate, do algebra, physics, chemistry, etc. But, even a brief exposure `sympy` is useful to you, because once you know that there is such a tool, you will think of it in the future when you have a relevant project. Nobody learns all of `numpy`, `scipy`, `sympy` (and other libraries of the Python scientific computing stack) all at once. We learn them piecemeal on a need-be basis.

Recall that the three main modules of a DE are 1) parser; 2) differentiation rules applier (DRA); and 3) lambdifer. Thus, you will type and save your code for Problems 01 and 02 in `lambdifier.py` and `dra.py` included in the zip. Included in the zip is `cs3430_s24_hw05_uts.py` where I have written

a few unit tests you can use to test your code with as you implement it. You can 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.

Also included in the zip are all the auxiliary files, i.e., `var.py`, `const.py`, `pwr.py`, `prod.py`, `plus.py`, `maker.py`, and `tiny_de.py`.

You will code up your solutions to Problem 03 in `nra.py`, i.e., the Newton-Raphson part of the assignment.

Coding Lab: Part 1

Install sympy (cf. www.sympy.org) and run `sympy_tests.py` included in the zip. We went over some differentiation and lambdification code segments in this file in Lecture 11 (cf. the PDF of this lecture in Canvas and/or your class notes). We will use `sympy` in the unit tests to compare the results of our DE.

Below is a code fragment from `sympy_tests.py` where we 1) define a symbol (symbols are more or less equivalent to our variable objects), 2) define a polynomial function object that uses that symbol, (the object is an instance of the sympy `Add` class (cf. our `plus` class)), 3) differentiate the polynomial, 4) lambdify it, i.e., turn it into a Python function, 5) evaluate the Python function at 1, 6) lambdify the 1st derivative of the polynomial, and 7) evaluate evaluate the obtained function also at 1.

```
### 1) define x to be a Symbol
x = symbols('x')
assert isinstance(x, Symbol)
### 2) define a polynomial object
f = 10.0*x**3 - 5*x**2 + 10.0*x + 500.0
### next line prints out 'f = 10.0*x**3 - 5*x**2 + 10.0*x + 500.0'
print('f = {}'.format(f))
assert isinstance(f, Add)
### 3) let's compute 1st derivative
df1 = f.diff()
### the next line prints out 'df1 = 30.0*x**2 - 10*x + 10.0'
print('df1 = {}'.format(df1))
assert isinstance(df1, Add)
### 4) let's lambdify the original polynomial
lf = lambdify((x), f)
# lf(1) = 10.0*1**3 - 5*1**2 + 10.0*1 + 500.0 = 515
### 5) let's evaluate the poly at 1.
assert np.allclose(lf(1), 515)
### 6) let's lambdify 1st derivative of the polynomial
ldf1 = lambdify((x), df1)
### ldf1(1) = 30.0*1**2 - 10*1 + 10.0 = 30
### 7) let's evaluateion 1st derivative at 1
assert np.allclose(ldf1(1), 30)
```

Coding Lab: Part 2

Let us dive into the coding structure of the tiny DE whose parts you will implement in this assignment. The top class is defined in `tiny_de.py`. It imports the three main components, i.e., the parser, the lambdifier, and the DRA, and then calls appropriate methods of those components. You do not need to modify anything in `tiny_de.py`.

```
from poly_parser import poly_parser
from lambdifier  import lambdifier
from dra         import dra

class tiny_de(object):
    def parse(self, text):
        """parse text into a const/var/pwr/prod/plus object."""
        return poly_parser.parse_sum(text)

    def lambdify(self, ds):
        """convert a data structure ds to Py function."""
        return lambdifier.lambdify(ds)

    def diff(self, ds):
        """differentiate a data structure ds."""
        return dra.diff(ds)
```

The other three methods of `tiny_de` deal with text files that contain textual representations of polynomials, one polynomial per file.

```
    def parse_file(self, file_path):
        """
        return a list of data structures from polynimals in a file file_path.
        the file contains one polynomial text per line
        """
        with open(file_path, 'r') as inf:
            return [self.parse(ln) for ln in inf.readlines()
                    if len(ln) > 0]

    def lambdify_file(self, file_path):
        """
        returns a list of lambdified polynomials given in a file file_path.
        """
        with open(file_path, 'r') as inf:
            return [self.lambdify(self.parse(ln)) for ln in inf.readlines()
                    if len(ln) > 0]

    def lambdify_diff_file(self, file_path):
        """
        returns a list of lambdified differentiated polynomials in a file file_path.
        """
        with open(file_path, 'r') as inf:
            return [self.lambdify(self.diff(self.parse(ln))) for ln in inf.readlines()
                    if len(ln) > 0]
```

I included the file `polys.txt` with several sample polynomial texts with the syntactic conventions we discussed in the previous 2 lectures.

```
5x^2
3x^2 + 10x^1
10x^3 + 3x^2 + -2x^1 + -10x^0
-14.13x^5 + 10.1x^4 + 7x^3 + 11.11x^2 + 5x^1 + -12x^0
13x^4 + 10x^3 + -7x^2 + 11x^1 + -9x^0
```

Recall from the last three lectures that we use only `'+'` to connect the summands of the polynomial. If the coefficient of a specific summand is negative, we put `'-'` right before the coefficient. Each summand is in the form of a real, a variable (we will use only `'x'`), the caret `'^'`, and another real that specifies the degree of the variable. The text `'x^2'` parses into a `pwr` object whose base is the variable with the name of `'x'` and whose degree is the constant with the value of 2. The text `'5x^2'` parses into a product object whose first multiplicand is a `constant` whose value is 5 and whose section multiplication is a `pwr`.

Let us import all these classes and `maker.py`. Recall that `maker` is a factory class. It is a single point of entry into our object manufacturing process. In other words, whenever we want to make an object, we use `maker`.

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

Here's how we make variables.

```
>>> 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
```

We move on to a few constant objects. All constants have float values.

```
>>> c1 = maker.make_const(1.0)
>>> c2 = maker.make_const(-3.0)
>>> import math
>>> c3 = maker.make_const(math.sqrt(10))
>>> isinstance(c1, const)
True
>>> isinstance(c2, const)
True
>>> isinstance(c3, const)
True
>>> import numpy as np
>>> np.allclose(c1.get_val(), 1.0)
True
>>> np.allclose(c2.get_val(), -3.0)
True
>>> np.allclose(c3.get_val(), math.sqrt(10))
True
```

Let's make a few `pwr` objects.

```
>>> 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)
>>> np.allclose(p1.get_deg().get_val(),2.0)
True
>>> np.allclose(p2.get_deg().get_val(),2.0)
True
>>> np.allclose(p3.get_deg().get_val(),5.0)
True
```

A polynomial text parses into a `plus` object that consistss. The class that allows us to represent sums is in `plus.py`. By the way, we do not use the name `sum`, because it's the name of a Python function. Let's make us a few binary sum objects. In the most basic case, we parse a summand by using `poly_parser.parse_elt()`

```
>>> elt = poly_parser.parse_elt('5x^2')
>>> print(elt)
(5.0*(x^2.0))
>>> isinstance(elt.get_mult1(), const)
True
>>> np.allclose(elt.get_mult1().get_val(), 5)
True
>>> isinstance(elt.get_mult2(), pwr)
True
>>> isinstance(elt.get_mult2().get_base(), var)
True
>>> elt.get_mult2().get_base().get_name() == 'x'
True
>>> isinstance(elt.get_mult2().get_deg(), const)
True
>>> np.allclose(elt.get_mult2().get_deg().get_val(), 2)
True
```

We represent polynomials with arbirarily many binary sums. A binary sum has exactly 2 summands referred to as element 1 (elt1) and element 2 (elt2). In the simplest case, a polynomial is just one product, e.g., `'5x^2'` or `'10x^-2'`. Then, we can build a polynomial out of 2 product objects. If we need 3 product objects in a polynomial, we use one binary sum for the first two product objects to

make a binary sum and then use another binary sum with the first binary sum and the third product object. Then, inductively, on to 4 products, 5, etc. For example, here's how we can represent Here's how we can construct manually $x^2 + x + 10$.

```
>>> prod1 = maker.make_prod(maker.make_const(1.0), maker.make_pwr('x', 2.0))
>>> prod2 = maker.make_prod(maker.make_const(1.0), maker.make_pwr('x', 1.0))
>>> prod3 = maker.make_prod(maker.make_const(10.0), maker.make_pwr('x', 0.0))
>>> plus1 = maker.make_plus(maker.make_plus(prod1, prod2), prod3)
>>> print(prod1)
(1.0*(x^2.0))
>>> print(prod2)
(1.0*(x^1.0))
>>> prod3 = maker.make_prod(maker.make_const(10.0), maker.make_pwr('x', 0.0))
>>> print(prod3)
(10.0*(x^0.0))
>>> plus1 = maker.make_plus(maker.make_plus(prod1, prod2), prod3)
>>> print(plus1)
(((1.0*(x^2.0))+(1.0*(x^1.0)))+(10.0*(x^0.0)))
```

Note that we made everything explicit in our representation of the polynomial, i.e., we represented x^2 as `'1x^2'`, x as `'1x^2'`, and 10 as `'10x^0`. The above code segments show what is going on under the hood of the parser when we do the following.

```
>>> isinstance(p2, plus)
True
>>> elt1 = p2.get_elt1()
>>> elt2 = p2.get_elt2()
>>> print(elt1)
((5.0*(x^2.0))+(-10.0*(x^5.0)))
>>> isinstance(elt1, plus)
True
>>> print(elt2)
(13.0*(x^-2.0))
>>> isinstance(elt2, prod)
True
```

Let's use `poly_parser` create the data structure to represent $5x^2 - 10x^5 + 13.3x^-2$.

```
>>> p2 = poly_parser.parse_sum('5x^2 + -10x^5 + 13.x^-2')
>>> print(p2)
>>> print(p2)
(((5.0*(x^2.0))+(-10.0*(x^5.0)))+(13.0*(x^-2.0)))
```

Here's how we can get various components of `p2`.

```
>>> elt1 = p2.get_elt1()
>>> elt2 = p2.get_elt2()
>>> print(elt1)
((5.0*(x^2.0))+(-10.0*(x^5.0)))
>>> isinstance(elt1, plus)
True
>>> print(elt2)
(13.0*(x^-2.0))
```

```
>>> isinstance(elt2, prod)
True
>>> isinstance(elt1.get_elt1(), prod)
True
>>> isinstance(elt1.get_elt2(), prod)
True
>>> np.allclose(elt1.get_elt1().get_mult1().get_val(), 5)
True
>>> np.allclose(elt1.get_elt1().get_mult2().get_deg().get_val(), 2)
True
```

I wrote 8 detailed parser unit tests `cs3430_s24_hw05_uts.py`, `test_parser_ut01()`, ..., `test_parser_ut01()` that illustrate how the parser works and how we can access different elements of the parsed objects. Take a look at them to better understand the object-oriented design of the polynomial representation.

Problem 1 (1 point)

Implement the method `lambdifier.lambdify_plus()` in `lambdifier.py` that converts a data structure representing a binary sum into a Python function. Use the unit tests for this problem to test your solution. These tests use `sympy` to compare our lambidification with `sympy`'s.

Problem 2 (1 point)

Implement the method `dra.diff_pwr()` in `dra.py` that differentiates a `pwr` object. Use the unit tests for this problem to test your solution. These tests also use `sympy` to compare our differentiation with `sympy`'s.

Problem 3 (3 points)

Now we put our tiny DE to use. The file `nra.py` contains the stubs of two static methods you will implement to find zero roots of polynomials with the Newton-Raphson Algorithm (NRA) (cf Slides 20, 21, 22 in the Lecture 11 PDF and/or your class notes). Use the following methods of the `tiny_de` class in your implementation `parse(text)`, `diff(ds)`, and `lambdify(ds)`.

```
class nra(object):

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

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

The method `zr1(text, x0, num_iters=3)` takes a string `text` with a polynomial, the first approximation to a zero root `x0`, and the number of iterations. It runs the 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()`. The third argument specifies the difference between two consecutive zero root approximations. This

method keeps on computing these approximations until this difference is $\leq$ `delta`. This method is not as numerically stable as `zr1()`, because it compares floats, but it has its place.

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

```
def check_zr(text, zr):
    tde = tiny_de()
    f = tde.lambdify(tde.parse(text))
    return np.allclose(f(zr), 0.0)
```

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

The file `cs3430_s24_hw05_uts.py` contains several unit tests to test finding zero roots of different polynomials.

The Famous $\sqrt{2}$

I would 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_prob03_ut04(self):
        text = '1x^2 + -2x^0'
        ni = 7
        zr = nra.zr1(text, 1.0, num_iters=ni)
        print('\ntest_nra_ut04: zr={}; num_iters={}'.format(zr, ni))
        assert nra.check_zr(text, zr)
        assert np.allclose(math.sqrt(2), zr)
```

There is a story here, sad but inspiring. The ancient Greek mathematicians and astronomers (especially, the Pythagoreans) believed that the natural numbers (by the way, 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, i.e., did not contain any breaks in them. 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 them. 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 gave up 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). There is nothing crazy about irrational numbers.

The other unit tests use the NRA to approximate the value of $\sqrt{3}$, $\sqrt{5}$, $\sqrt{7}$, $\sqrt{11}$, $\sqrt{13}$, ..., and generalize with the unit test 09. Let's take a look at it.

```
def test_prob03_ut09(self):
    for i in (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71):
        text = '1x^2 + -{}x^0'.format(i)
        zr = nra.zr1(text, 1.0, num_iters=7)
        assert nra.check_zr(text, zr)
        assert np.allclose(math.sqrt(i), zr)
        print('\nour sqrt({}) = {}; math.sqrt({}) = {}'.format(i, zr, i, math.sqrt(i)))
```

It compares our implementation of NRA and tiny DE with the Python's `math.sqrt()` function. Here's what I get in Python 3.6.7 on Ubuntu 18.04 LTS.

```
our sqrt(2) = 1.4142135623730951; math.sqrt(2) = 1.4142135623730951
our sqrt(3) = 1.7320508075688772; math.sqrt(3) = 1.7320508075688772
our sqrt(5) = 2.23606797749979; math.sqrt(5) = 2.23606797749979
our sqrt(7) = 2.6457513110645907; math.sqrt(7) = 2.6457513110645907
our sqrt(11) = 3.3166247903554; math.sqrt(11) = 3.3166247903554
our sqrt(13) = 3.605551275463989; math.sqrt(13) = 3.605551275463989
our sqrt(17) = 4.123105625617661; math.sqrt(17) = 4.123105625617661
our sqrt(19) = 4.358898943540673; math.sqrt(19) = 4.358898943540674
our sqrt(23) = 4.79583152331272; math.sqrt(23) = 4.795831523312719
our sqrt(29) = 5.385164807134504; math.sqrt(29) = 5.385164807134504
our sqrt(31) = 5.567764362830022; math.sqrt(31) = 5.5677643628300215
our sqrt(37) = 6.08276253029822; math.sqrt(37) = 6.082762530298219
our sqrt(41) = 6.4031242374328485; math.sqrt(41) = 6.4031242374328485
our sqrt(43) = 6.557438524302001; math.sqrt(43) = 6.557438524302
our sqrt(47) = 6.855654600401045; math.sqrt(47) = 6.855654600401044
our sqrt(53) = 7.280109889280524; math.sqrt(53) = 7.280109889280518
```

```
our sqrt(61) = 7.81024967590673; math.sqrt(61) = 7.810249675906654
our sqrt(67) = 8.185352771872816; math.sqrt(67) = 8.18535277187245
our sqrt(71) = 8.426149773177292; math.sqrt(71) = 8.426149773176359
```

Let's take stock of what just happened. In just seven iterations of NRA based on our tiny DE, we got as good approximations as `math.sqrt()` gives us.

What To Submit

Submit your code in `lambdifier.py` and `dra.py` and `nra.py`. It will be easiest for me to run the unit tests on your code if you you place all the files (i.e., `var.py`, `const.py`, `pwr.py`, `plus.py`, `prod.py`, `maker.py`, `poly_parser.py`, `lambdifier.py`, `dra.py`, `tiny_de.py`, `nra.py`), and `polys.txt` into one directory, zip it into `hw05.zip`, and upload your zip in Canvas.

Happy Hacking!

References

1. L. Goldstein, D. Lay, D. Schneider, N. Asmar. *Calculus and its Applications.*
2. E. Burger. *Zero to Infinity: A History of Numbers.* The Great Courses Publishing, 2007.
3. www.sympy.org.