ENGN 2020:

Homework #4

Brown University School of Engineering

Assigned: March 4, 2019, Due: March 11, 2019

Problem 1

A powerful aspect of Python (and other object-oriented languages) is the ability to create new types of objects. For example, imagine if you had something that could create a quadratic function:

```
my_equation = Quadratic(a=3., b=2., c=8.)
y = my_equation.get_value(x=2.)
```

That is, the new object called `my_equation` is a quadratic equation that has stored values of the coefficients, and you can quickly call it without having to feed in the coefficients each time. Here, we'll look at how we could make `Quadratic`, which is called a *class* in Python:

```
class Quadratic:
    """Creates a quadratic equation. Initialize with the coefficients
    a, b, and c."""
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c
    def get_value(self, x):
        """Returns the value of the quadratic equation at x."""
        return self.a * x**2 + self.b * x + self.c
```

We'll describe the key aspects of how this works, but you should consult python documentation for rigorous details:

- The lines marked `def` are called *methods* of the class, and are functions that belong to the class. You can make as many methods as you like.
- The word `self` must be the first argument to each method. This object is just a common location to store variables and access other methods of the class. As seen, the coefficents `a`, `b`, and `c` are stored in `self` in `__init__` and retrieved from `self` in `get_value`.
- The class must have the method called `__init__`. (Note that it is two underscores before and after.) This is the function that is run when the class is first called; in our example above creating the object `my_equation`.

- By convention, classes are CamelCased, to make it easier to spot them when reading others' code (or your own code after a couple of weeks).

Your assignment is to create a class called `Polynomial`, that acts like the class above, but instead creates an nth-order polynomial when fed a list of coefficients. For example, initializing your class with `coefficients=[3, 2, 8]` would re-create the quadratic of the example above. You should use the below code as a starting point.

File attached here.

```python
class Polynomial:
    """Creates an nth-order polynomial, given a list of its coefficients."""
    def __init__(self, coefficients):
        self.coefficients = coefficients

    def get_value(self, x):
        """Returns the value of the polynomial equation at x."""
        # Write me.
```

Submission

Label: hw4_1
Points: 2
Class initialization variables: `coefficients` (a list of numbers)
Class method(s): `get_value` (which takes in the number `x` and returns the value of the polynomial, another number)

Problem 2

In factoring roots out from a function f (forming a new function g), it can quickly become cumbersome to derive and subsequently code functions for g and g'.

Part a. Assuming that you are factoring out n roots from f, derive general expressions for g and g'. (Hint: it can be useful to define the root-factoring equations as $f_i \equiv (x - r_i)^{-1}$. You can leave your answer in terms of f, f_i, and n.)

Paper Submission

(Turn in expressions for g and g'.)
Points: 1

Part b. Now, write a python *class* to make this systematic. [1] Your class should work as follows: it should take in two functions, `f` and `fprime`, as well as a list of `roots` that have already been

[1]If it's not clear to you what exactly a class is or how this could work, we will go over this aspect of python in class on Wednesday.

found. It should then be capable of returning the values of g and g'. That is, a script that calls your class might work as:

```python
def f(x):
    return x**3 - 6 * x**2 + 11. * x - 6.

def fprime(x):
    return 3 * x**2 - 12 * x + 11.

newfunction = RootFactor(f=f, fprime=fprime, roots=[1., 3.])

root = fsolve(func=newfunction.get_f, fprime=newfunction.get_fprime, x0=0.)
```

In this example, it should find the remaining root 2. You can also test your code on the previous homework problem. Note that the above script is identical to how you would normally write a script to call `fsolve`, but the use of a class allows you to modify this function in a single additional line.

Your assignment is to write this python class, filling in the blanks of the following:

File attached here.

```python
class RootFactor:
    """A class that allows for the creation of a new function g from a
    function f and a list of known roots."""

    def __init__(self, f, fprime, roots):
        self.f = f
        self.fprime = fprime
        self.roots = roots
        # Write me.

    def get_f(self, x):
        """Returns the value of g given a value x."""
        # Write me.

    def get_fprime(self, x):
        """Returns the value of the derivative of g at the value x."""
        # Write me.
```

Paper Submission

(Turn in a print-out of your code.)
Points: 1

Part c. Now, use your code to solve for roots of the equation:

$$f(x) = 0.2x^2 - 20\sin x + 30\cos(0.3x + 1)$$

Use the following skeleton of a script when you are ready to submit.

File attached here.

```python
from submission.client import submit

class RootFactor:
    # Your working code from part b here.

def get_f(x):
    # Write me.

def get_fprime(x):
    # Write me.

def use_my_code(roots, x0):
    newfunction = RootFactor(f=get_f, fprime=get_fprime, roots=roots)
    root = fsolve(func=newfunction.get_f, fprime=newfunction.get_fprime,
                  x0=x0)
    return root

submit(use_my_code, assignment='hw4_4c')
```

Submission

Label: hw4_2c
Points: 2
Input variables: `roots` (a list of numbers), `x0` (a number)
Output: a number

Problem 3

Consider the system of equations $\underline{\mathbf{f}} = [f_1 \;\; f_2]^{\mathrm{T}} = \underline{\mathbf{0}}$ with unknown variables $[x_1 \;\; x_2]^{\mathrm{T}}$, where

$$\begin{aligned} f_1(x_1, x_2; \theta) &= \sin 3x_1 - x_2 \\ f_2(x_1, x_2; \theta) &= x_1^2 - x_2 - 3x_1 + \theta \end{aligned}$$

Part a. Identify all the bifurcation points $\theta = \theta^*$ at which the number of real roots changes. (Give precise numerical answers.)

Paper Submission

(Turn in a PDF to Canvas.)
Points: 1

Part b. At what particular value of θ does the system have exactly one real root? (Hint: you may try a graphical procedure to choose the correct value of θ^*, solving each equation for x_2 and plotting the resulting functions on the x_1–x_2 plane for various values of θ.)

Paper Submission

(Turn in a PDF to Canvas.)
Points: 1

Part c. What is the maximum number of real roots this system exhibits at any value of θ?

Paper Submission

(Turn in a PDF to Canvas.)
Points: 1

Part d. In what range or ranges of θ does this system exhibit its maximum number of real roots?

Paper Submission

(Turn in a PDF to Canvas.)
Points: 1

Problem 4

Here, we will examine the behavior of the Nelder–Mead algorithm.

Part a. Use the "contour" (or "contourf") function of pyplot to plot the below function, choosing bounds on x_1 and x_2 and suitable contour density such that the local minima of the function are clearly visible.

$$f(x_1, x_2) = 0.045x_1^4 - x_1^2 + 0.5x_1 + 0.065x_2^4 - x_2^2 + 0.5x_2 + 0.3x_1x_2$$

Paper Submission

(Turn in a PDF to Canvas.)
Points: 1

Part b. Write a function that performs a single step of the Nelder–Mead algorithm, as described on wikipedia.[2] Your Nelder–Mead function should take in the variables `f` and `vertices`, where `f` is a reference to any function that you would like to be optimized and `vertices` is a list of the points that make up the vertices of the simplex. Your function should return the new list of vertices after this single step. Use wikipedia's standard values for the method's parameters.

To submit this to the autograder, use the following wrapper function, with your function, as follows:

[2] `https://en.wikipedia.org/wiki/Nelder%E2%80%93Mead_method#One_possible_variation_of_the_NM_algorithm`

```
def get_f(x1, x2):
    return (0.045 * x1**4 - x1**2 + 0.5 * x1
            + 0.065 * x2**4 - x2**2 + 0.5 * x2
            + 0.3 * x1 * x2)

def wrapper(vertices):
    """This is just used to submit."""
    return step_neldner(f=get_f, vertices=vertices)

submit(wrapper, 'hw4_4b')
```

(In the above, `step_neldner` is the name of your function; change this is you called it something else.)

Submission

Label: hw4_4b
Points: 2
Input variables: `vertices` (a 3×2 numpy array containing the x_1, x_2 vertices as rows)
Output: a 3×2 numpy array containing the new vertices

Part c. Now, run this algorithm by making an initial simplex guess and repeatedly calling your step function until you are confident it has converged to a local minimum. Plot the triangles of each iteration on top of the contour plot you generated in part a.

Paper Submission

(Turn in a PDF to Canvas.)
Points: 1