CSCI-B 505 Applied Algorithms (3 cr.) № 1

Dr. H. Kurban
Computer Science
School of Informatics, Computing, and Engineering

Indiana University, Bloomington, IN, USA

August 30, 2021

Contents

Introduction

This homework is meant as precursor to the course: developing working knowledge of Python and being exposed to some of the concepts of *Applied Algorithms* as we move through the semester. Because of the consequences of Covid, these first few homeworks will not be graded; rather, they are to help you prepare. The solutions are given at the end, but as graduate students you know that skipping to the end without trying to solve the problems makes the exercise unhelpful. We urge you, in the strongest terms, to treat these as though they were homework and learn from them. With respect to this, we are not giving you the code explicitly–you should type it in–this will help you immensely, since you'll have bugs and bugs are always gifts to better understand code! You are also encouraged to explore Python beyond what we present here–we simply do not have time to teach the language completely–as computer and data scientists we know that *if you know one language, you know them all*. We list here the salient topics for this particular homework:

- Writing and executing a Python program
- Using both named and unnamed functions and modules
- Using arrays
- loops: bounded, unbounded, recursion
- Visualization of data

We end with three problems drawn from content above. Solutions are presented at the end. As graduate students you are aware that there are often no single correct answer; rather, collections of correct answers. If you have found a solution that's different from what we present, that's great! The important point is to make sure you understand the underlying concept. If you're unsure, contact us.

Problem 1:Writing and Executing a Python Programs

Python is now arguably among the most popular programming languages. While it has shortcomings, as all languages do, its strengths of community, ease of programming, modules (libraries), industry support, incredible visualization modules, continual evolution (to list a few), make it an appealing language to study applied algorithms. In our examples, we will often switch from Windows, Apple OS, and Linux – the OS doesn't matter at this point–the commands are essentially identical for what we will be doing.

Getting Started

1. Install Python (www.python.org)

2. (optional) Install an editor – you can write your code in any ASCII editor–for this example, we're using IDLE.

3. Create a folder for the class. We are using `505` as our folder name. In this folder, create a subfolder `Assignment_1`.

4. Open an editor and write this code in folder `Assignment_1`

   ```python
   print("Hello, Indiana University!")
   ```

   and save it as `hello_IU.py`.

5. Open a shell and execute the Python:

   ```
   PS D:\505> py .\Assignment_1\hello_IU.py
   Hello, Indiana University!
   PS D:\505>
   ```

   On our machine we abbreviated python to py. You do not need to do this. The code is executed.

Adding Modules

In this problem, we will add the `random` module (please read about this) and our own. As you probably know, deep learning relies on threshold functions: if a value is less than, say, zero, it's returns -1; otherwise it returns 1. We will write this as our own module. We will use the `random` module to generate data.

1. Inside `Assignment_1`, write this program:

   ```python
   def f(x):
       if (x >= 0):
   ```

```
        return 1
    else:
        return -1
```

and name it `neuron.py`. Make sure you understand what it does.

2. Inside `Assignment_1`, write this program:

```
import neuron
import random as rn

for _ in range(5):
    data = rn.randint(-10,10)
    print("{0:>4} {1:>4}".format(data, neuron.f(data)))
```

and name it `threshold.py`. Make sure you understand what it does.

3. We have written `threshold.py` so that it uses `neuron` and `random`. Execute the program:

```
PS D:\505> py .\Assignment_1\threshold.py
   8    1
  -2   -1
   9    1
  -9   -1
   7    1
PS D:\505>
```

Observe we used formatted output to make readability easier.

Problem 2: Named and Unnamed Functions

Python treats functions as first-class objects. This comes in handy. Newton's difference quotient describes the derivative:

$$f'(x) \quad = \quad \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{1}$$

Observe that Eq. 1 is a function! There are times when we need to compute a derivative without knowing the function *a priori*. We can approximate a derivative:

$$f'(x) \quad \approx \quad \frac{f(x+h) - f(x-h)}{2h} \tag{2}$$

for very small values of h. Like the equation before, Eq. 2 is a function.

For example, let $f(x) = x^3/10$. Let's find the value of the derivative at $x = 2.5$.

$$f(x) \quad = \quad x^3/10 \tag{3}$$

$$f'(x) \quad = \quad (\tfrac{3}{10})x^2 \tag{4}$$

$$f'(2.5) \quad = \quad (\tfrac{3}{10})2.5^2 = 1.875 \tag{5}$$

We will write a program that inputs univariate function of x and value and displays the derivative at 2.5.

1. Inside `Assignment_1`, write this program:

```
1 fn = input("input function: ")
2 fn = eval("lambda x: " + fn)
3 val = float(input("value: "))
4
5 def derivative(f):
6     h = 0.00001
7     return lambda x: (f(x + h) - f(x - h))/(2*h)
8
9 print(derivative(fn)(val))
```

and name it `derivative.py`. A couple of observations: Python's I/O is *via* strings. So you must change the type (when inputting) if you're not going to use the string type. The `eval` function converts the text to a valid Python expression. We changed the string 2.5 to float on line 3. The derivative function returns a function described by Eq. 2. Here is a run of the program:

```
PS D:\505> py .\Assignment_1\derivative.py
input function: (1/10)*(x**3)
value: 2.5
1.8750000000178344
PS D:\505>
```

As you can obviously see, this is an approximation.

Problem 3: Arrays

Python's own array type isn't considered as good as the array in the array module. We will only use the numpy module. A array is a contiquous portion of memory allocated for a type. Since applied algorithms have a great of data as arrays, it's important you become familiar with them. Arrays are collections–each element is accessed by an index. Python allows slicing which will we not cover here, but you should look into it.

There will be *many* "quick tricks" to do operations on arrays–Python itself has many. You should be cognizant, however, that these are only run-time fixes–the underlying algorithm is not changed.

1. Inside `Assignment_1`, write this program:

```python
import numpy as np
import random as rn

def search1(v, d_array):
    for i in range(0, d_array.shape[0]):
        if v == d_array[i]:
            print("found {0} at {1}".format(v,i))

def search2(v, d_array):
    for i,j in enumerate(d_array):
        if v == j:
            print("found {0} at {1}".format(v,i))

def search3(v, d_array):
        if np.any(d_array[:] == v):
            print("found")

data = [0,1,10,2,1,23,1]

da1 = np.array(data)
da2 = np.empty(7)
da3 = np.empty((2,3))

print(data)
print(da2)
da2 = da1
print(da2)
da2[0] = -1000
print(da1)
print(da3)

for i in range(da3.shape[0]):
    for j in range(da3.shape[1]):
        da3[i][j] = rn.randint(-5,5)

print(da3)

search1(23,da1)
search2(23,da1)
search3(23,da1)
```

and name it `array1.py`.

2. Execute `array1.py`

```
PS D:\505> py '.\Assignment_1\array1.py'
[0, 1, 10, 2, 1, 23, 1]
[3.76624467e+112 1.33501124e+151 8.71458132e+150 6.52933223e+044
 2.43395092e-152 1.14407192e+243 0.00000000e+000]
[ 0  1 10  2  1 23  1]
[-1000     1    10     2     1    23     1]
[[6.23042070e-307 1.16824187e-307 1.89146896e-307]
 [1.02360528e-306 1.33511290e-306 1.03980870e-312]]
[[-4.  1.  0.]
 [ 0. -5.  1.]]
found 23 at 5
found 23 at 5
found
```

Observe that when printing, a list includes commas, whereas the numpy array does not. Also observe that arrays are initialized with numbers albeit small. Also observe that arrays are call-by-reference–so the assignment of `da2` and subsequent altering of value changes `da1`. We can avoid this explicitly–you should read how. There are a number of ways to search through the array–you should always use simply scanning, since this is true to algorithms. Again, any fast tricks you see have other algorithms hidden–so, ultimately, it won't make a difference.

Problem 4: Looping

There are four ways to loop:

- Goto (which we will not ever use)
- Bounded looping (a fixed number of iterations)
- Unbounded looping (a potentially infinite number of iterations)
- Recursion (a potentially infinite number of iterations)

Bounded is strictly less powerful than either unbounded or recursion (which are equal in expressiveness). Some problems are more easily solved with one kind of looping than others. In Python, there is a limit to the stack when recursing. Also, tail-recursion is not treated differently than recursion. Here is an example we've sure you've seen:

$$f(0) = 0 \tag{6}$$

$$f(n) = n \times f(n), \quad n = 0, 1, 2, \ldots \tag{7}$$

We'll write this in three different forms.

1. Inside `Assignment_1`, write this program:

```python
def f(n):
    if n == 0:
        return 1
    else:
        return n * f(n-1)

def f1(n, acc=1):
    if n == 0:
        return acc
    else:
        return f1(n-1, acc*n)

def f3(n):
    v = 1
    for i in range(n,0,-1):
        v *= i
    return v

print(f(5), f1(5), f3(5))

print(f3(2021))
```

   and name it `recur.py`.

2. Execute the program:

```
PS D:\505> py '.\Assignment_1\recur.py'
120 120 120
78030193964181096707498359348000401057239437703732087033993257472396667↩
0638272179141117802934475394226002164644216644923930930542069974053352↩
9828443589112978094315790589459115326726585869546644590908033098153851↩
```

```
(a lot of numbers) ... 0
PS D:\505>
```

3. Replace the last print with `print(f(2021))` and execute the program. You'll see this:

```
120 120 120
Traceback (most recent call last):
  File ".\Assignment_1\recur.py", line 25, in <module>
    print(f(2021))
  File ".\Assignment_1\recur.py", line 5, in f
    return n * f(n-1)
  File ".\Assignment_1\recur.py", line 5, in f
    return n * f(n-1)
  File ".\Assignment_1\recur.py", line 5, in f
    return n * f(n-1)
  [Previous line repeated 995 more times]
  File ".\Assignment_1\recur.py", line 2, in f
    if n == 0:
RecursionError: maximum recursion depth exceeded in comparison
```

As you can see, Python has a strict limit on the stack. This should *not* pose any problem for us during the semester in general. You should remember any recursive program can be written using only unbounded loops–though it might be difficult and even tedious.

Problem 5: Visualization

One of the most heavily used elements of Python is its visualization. In this example we plot the curve in Problem 2.

1. Inside `Assignment_1`, write this program:

```python
import matplotlib.pyplot as plt
import numpy as np

data = np.linspace(-10,10,100)
fn = (1/10)*(data**3)


fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

plt.title('$f(x) = (1/10)x^3$')
plt.plot(data,fn, 'r')
plt.show()
```

The output is shown in Fig. 1.

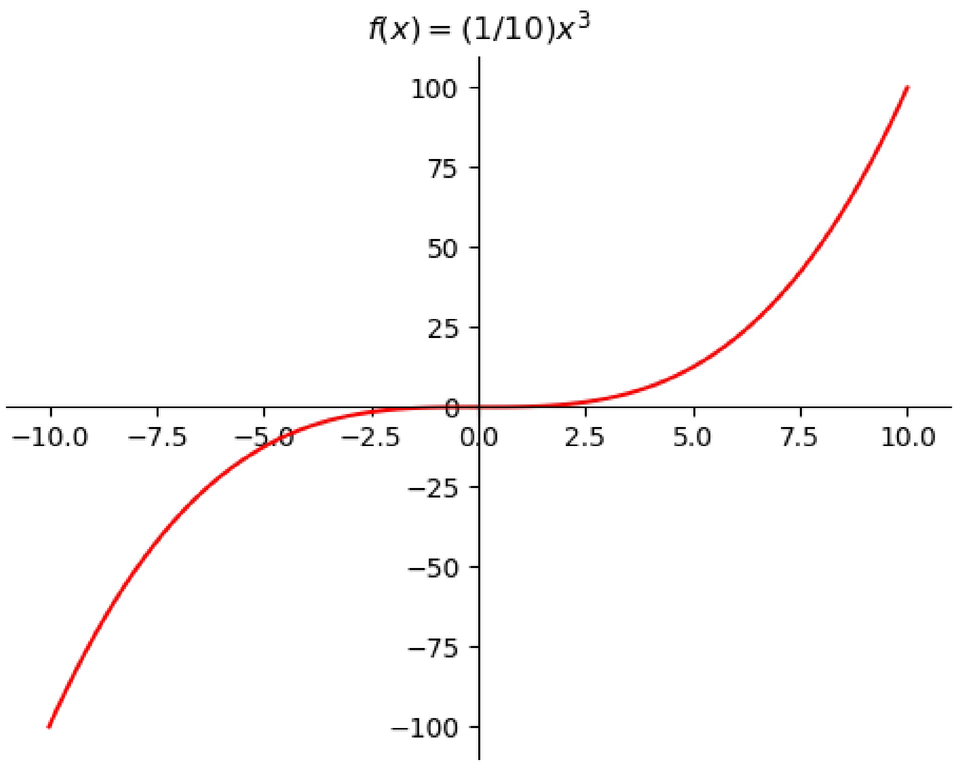

Figure 1: Visualizing f

Problem 6: Practice

1. The second derivative is the derivative of the first derivative. Let's look out original example:

   For example, let $f(x) = x^3/10$. Let's find the value of the 2^{nd} derivative at $x = 2.5$.

$$f(x) \quad = \quad x^3/10 \tag{8}$$

$$f'(x) \quad = \quad (\tfrac{3}{10})x^2 \tag{9}$$

$$f''(x) \quad = \quad (\tfrac{2\times3)}{10}x = (\tfrac{3}{5})x \tag{10}$$

The value of $f''(2.5) = 1.5$. Add a new function called der2 that takes a function and returns the second derivative. Verify with the value 2.5.

2. Parametric problems are those for which we have (or assume to have) an underlying distribution. Two statistics we are always interested in (with context) are the mean and variance. Assume we have a data set of non-negative integers $|D| = n$. The mean (a measure of central tendency) is:

$$\mu = \frac{1}{n}\sum_{d \in D} d \tag{11}$$

The variance is spread around μ and is:

$$\sigma^2 = \frac{1}{n}\sum_{d \in D}(d - \mu)^2 \tag{12}$$

Write a program that generates 100 random integers from 0 to 100 inclusive (repeating) and store them into an array. Find the mean and variance. The mean and variance should be written as functions. Look-up the size attribute of numpy arrays before you begin.

3. This problem will take some thinking–it will help you understand the array type. In the original plot, we created a second array implicitly on line 5. Python understood we had an entire array as input and then created an array of values each applied to the values in the initial array. For this problem, we'll have to eschew this and create the arrays explicitly. What we want to do is draw our original curve, but include the first derivative curve as well. The challenge is that our derivative function is not compatiable with the array. So we need to explicitly create two arrays, one with f (red) and another with f' (blue). See how far you go. The output is in Fig. 2.

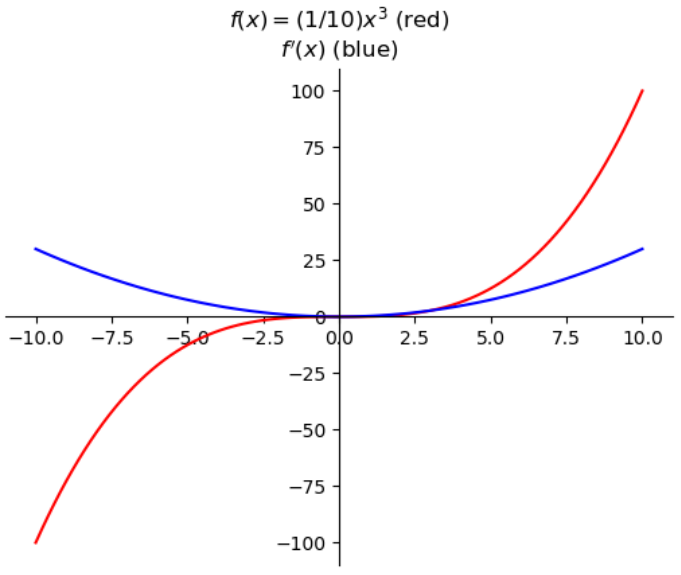

Figure 2: Visualizing f and its derivative.

Solutions

1. Here is a program

```python
fn = input("input function: ")
fn = eval("lambda x: " + fn)
val = float(input("value: "))

def der2(f):
    return derivative(derivative(f))

def derivative(f):
    h = 0.00001
    return lambda x: (f(x + h) - f(x - h))/(2*h)

print(derivative(fn)(val))

print(der2(fn)(val))
```

   and here is a run:

```
PS D:\505> py .\Assignment_1\derivative2.py
input function: (1/10)*(x**3)
value: 2.5
1.8750000000178344
1.5000006792220686
PS D:\505>
```

2. Here is a program:

```python
import numpy as np
import random as rn


def variance(data):
    mu_ = mu(data)
    n = data.size
    sum = 0
    for i in data:
        sum += (i - mu_)**2
    return sum / n

def mu(data):
    n = data.size
    sum = 0
    for i in data:
        sum += i
    return sum / n



asize = 100
da2 = np.empty(asize)
for i in range(asize):
    da2[i] = rn.randint(0,asize)

print("There are {0} integers.".format(da2.size))
print(mu(da2))
print(variance(da2))
```

and here's is the output of a run:

```
PS D:\505> py '.\Assignment_1\array2.py'
There are 100 integers.
48.09
994.1819
PS D:\505>
```

3. Here is a program:

```
import matplotlib.pyplot as plt
import numpy as np


def derivative(f):
    h = 0.00001
    return lambda x: (f(x + h) - f(x - h))/(2*h)

def f(x):
    return (1/10)*(x**3)


data = np.linspace(-10,10,100)

f_orig = np.empty(100)
for i,d in enumerate(data):
    f_orig[i] = f(d)

fp = np.empty(100)
for i,d in enumerate(data):
```

```python
    fp[i] = derivative(f)(d)



fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

plt.title(r'$f(x) = (1/10)x^3$ (red)' '\n' r"$f'(x)$ (blue)")
plt.plot(data,f_orig, 'r')
plt.plot(data,fp,'b')

plt.show()
```