NE 336: Lab 1 Group 1

Pendar Mahmoudi

The first section of this document is for information, if you feel confident enough to start the tasks, feel free to skip to section 2.

1 Concepts

1.1 Errors

A invaluable skill when learning programming is not to try to avoid error (no one is born a programmer except in movies) but to know what to do with error and how to handle it.

First we need to know what kinds of error there are.

1.1.1 Numerical Error

This is inherent to numerical methods and cannot be eliminated.

It is not error in the sense of a "mistake" but in the sense that numerical methods are approximate and therefore have some error between the approximation and the true value. so just because your code is working (hooray?) and giving your results within the criteria you defined, this does not mean your solution is without fault or absolutely correct.

There are two types of numerical errors.

1. Roundoff error – due to the way computers represent numbers.
2. Truncation error – due to the use of approximate rather than exact calculus.

1.1.2 Programming Error

These are the ones you will encounter when writing your programs. The more you practice programming, the better you will become at recognizing and eliminating these errors.

1. Syntax error – violation of the rules of the programming language. Very easy to make, but also fairly easy to fix
2. Semantic errors – the algorithm runs but does not do what was intended. These are relatively hard to find and fix.
3. Link/build or compile-time errors – errors when the program gets compiled. Commonly this might be a reference to a missing function or missing library. This should not be a concern in this course.
4. Runtime errors – errors that occur at the time a program is run. For example, missing or unexpected inputs.

Note : This assumes that you have the math correct. You might also have an error there, but that is not really a programming error...

Example
If I have the following function in my path

```
def my_factorial(n):
   # my factorial returns the factorial of n
   out=0
   for i in range(1,n+1):
      out=out*i
   return out
```

and try to evaluate

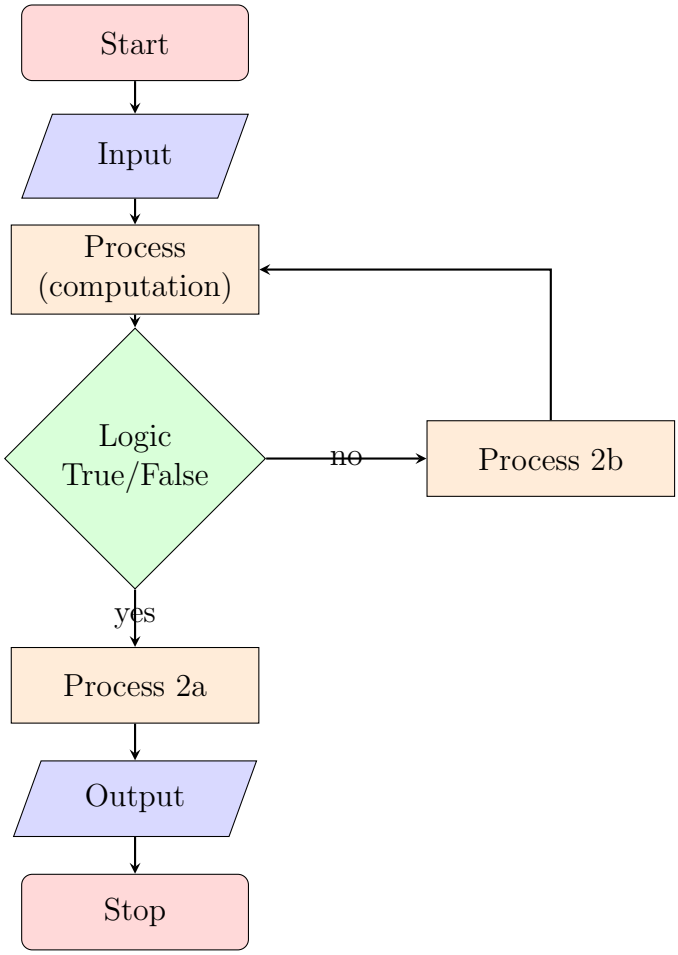
```
my_factorial(4)
```

What type of error do I have?

1.1.3 Flowcharts

You might have seen these before. Flowcharts are a great tool to lay out the idea for what the program should accomplish, before actually writing any code. The more you write out ideas on paper before implementing them, the less programming errors you'll have to deal with (trust me on this!).

There are a limited set of symbols used in flowcharts as seen in the example below.



1.2 Good Programming Practices

1. Each function should have documentation to describe its name, purpose (including inputs and outputs) and logic.

2. Write as many comments as you can!

   (a) These should be meaningful. Examples include describing variables as they are initialized, stating where values came from (references), ...

(b) Comments should be written such that future you can understand the code by looking at it months/years later from these comments and lines.

(c) If collaborating with others, commenting becomes indispensable to the success of your work.

3. Test your code against known results.

   (a) Think of various scenarios to test for your code. With experience, these will readily come to mind.

   (b) Use your ideas to test how the program will behave for both common and corner cases.

2 How to submit your answers to the lab

Please keep these points in mind as you go on to complete the lab questions.

- When working on these questions, please ask for help from the TAs and myself as needed.
- Discussion with classmates is encouraged but takings somebody else's code and submitting their work as your own is not acceptable. Please ask for clarification on this item if unsure.
- In terms of formatting your submission:
  - If the question is asking for a script or function, please submit as a python file.
  - If asked for analysis, you may include this as comments in your python file, handwritten work or however else you wish.
- Please include your name and student ID in each file that you submit.

3 Today's Lab

3.1 Numpy Polynomial module

The file called `np_poly_howto.py` provides a short and brief introduction to the numpy polynomial module. See `numpy.polynomial` for full documentation. You can use functions from this module to test your code.

3.2 Horner's scheme

3.2.1 Notation

In the NumPy polynomial module, the convention to represent a polynomial $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$ is $[a_0,\ a_1,\ a_2,\ \ldots,\ a_{n-1},\ a_n]$. Please note that an easy way to remember this is that coefficient at index i is for the term of degree i. For this exercise, you should adopt this convention.

Example : $2x^3 + 4x - 8$ would be $[-8,\ 4,\ 0,\ 2]$.

3.2.2 The method

The most obvious way to evaluate a polynomial is to take each coefficient in turn and then multiply by x to the appropriate power. This mirrors how you would type the problem into your calculator. For high-order polynomials, this requires a lot of multiplications. Multiplications are expensive (in a computational sense) and this will also allow round-off errors to accumulate. For example, $f(x) = 10x^5 + 8x^4 + 6x^3 + 4x^2 + 2x + 1$ would require $5 + 4 + 3 + 2 + 1 = 15$ multiplications and 5 additions. If for some reason you had a 100th order polynomial, you would have to do around 5000 multiplications.

Can we do improve on this? Specifically, by "improving" I mean doing the same calculation using fewer operations (multiplications and additions). Let's look at the polynomial above as an example. We need to calculate x^5, but we also need to find x^4. So, can we find x^4 first and then just do one extra multiplication to get x^5? The answer should clearly be yes; here is a scheme exploiting this idea.

Take x and multiply by the coefficient of the largest power (here, the 10 from the x^5 term):

$x(10)$ which multiples out to give	$10x$
Add the next coefficient to the result	
$(x(10)+8))$	$10x+8$
and again multiply everything by x:	
$x(x(10)+8)$	$10x^2+8x$
Repeat, adding and multiplying each time:	
$x(x(x(10)+8)+6)$	$10x^3+8x^2+6x$
$x(x(x(x(10)+8)+6)+4)$	$10x^4+8x^3+6x^2+4x$
$x(x(x(x(x(10)+8)+6)+4)+2)$	$10x^5+8x^4+6x^3+4x^2+2x$
Finally, add the constant term	
$x(x(x(x(x(10)+8)+6)+4)+2)+1$	$10x^5+8x^4+6x^3+4x^2+2x+1$

If you multiply out, you recover the original equation. This is called Horner's scheme and it can be shown that it is optimal in that it has the least number of multiplications for a general polynomial (it requires n multiplications and n additions; the naive method we started with can be shown to be the worst-possible effort, with $1/2(n^2+n)$ multiplications and n additions). The point is that the implementation scheme can affect the computational effort; consequently, it may also have implications for round-off error accumulation. To write your function, start with the line

```
def mypolyval(p,x):

    '''
    (list,float)-> float

    function to evaluate a polynomial p at x.
    '''
```

You should represent a polynomial in the same way that the NumPy does so you can test your code against the builtin methods easily!

3.3 Tasks for Horner's Scheme

3.3.1 Task 1 : Optional Bonus!

Write a flowchart to implement Horner's Scheme.

3.3.2 Task 2

Write a function called `mypolyval` that will accept an input polynomial and a value of x and return the value of that polynomial calculated using Horner's scheme. Put this in a module called `scheme.py`.

3.3.3 Task 3

Test your `mypolyval` function for a few polynomials (use Python's built-in functions for the "correct" values and check that both methods agree). As an aside, you could make these tests only run when `scheme.py` is run as a script by using `__name__` but this is optional.

Considerations

In terms of good programming practices, what can you test for the mypolyval(p,x) function?

For example, how does your polynomial function behave with the following inputs:

1. Various values for x:
   (a) positive, negative
   (b) real, imaginary
   (c) x as a vector or matrix
2. Different polynomials p:
   (a) p as a scalar
   (b) leading zeros

3.4 The Bisection Method

As an optional first step, you may want to develop a flowchart for this problem. A few notes on this:

- This is a design problem, so you probably need to iterate to come up with a final flowchart – it is hard to write it down correctly and clearly on the first attempt. Note that this is one of the main uses of the flow chart – it is easier to make these changes on paper than in the code.
- You may also refine the flowchart as you go (for example, what is the convergence test? how are errors handled?). We also talked about where the algorithm "fails" due to the way it is implemented – for example, if we happen to hit exactly on a root. You can either accept this as a "feature" or check for that condition and deal with it.
- The "right answer" is not unique.
- You can include more or less detail in the flowchart depending on what you want it for. The important part is that you provide enough understanding of the algorithm to translate it into code. In fact, you could start with a simpler (higher level) version and make it more detailed later on.

3.4.1 Translating the Flowchart into Code

We now need to translate the flowchart above into code. This means that we'll need to make some decisions about what variables are required, what to name them, etc. There are also different options for implementing the same logic. You need to know how to implement loops and conditional control in Python.

Here is the outline of the function

```
def bisection(polyf, xl, xu):
    '''
    This function finds a root xroot for the function polyf
    between xl and xu using the bisection method.
```

```
    inputs : polyf polynomial to find roots for
             xl    lower bound of interval
             xu    upper bound of interval
    output : xroot root of the function
    '''

    return xroot
```

The output of our function is just a number, called xroot. The input is a function and the two limits on an interval; how these should be represented is not specified. We have chosen two variables xl and xu for the limits. To represent the polynomial – let's use a row vector of coefficients,

x^2–$4x + 2 = 0$ is represented as [2 -4 1]

$-x^3 + 32x^2$–$1.1 = 0$ is represented as [-1.1 0 32 -1]

Now we have the inputs, you need to write the function. Here are some notes to keep in mind.

3.4.2 Initializing variables

First, we will declare the variables we intend to use. This makes the program easier for someone else to read, since they can look to see what each variable represents.

Use descriptive names for variables – they can include letters and numbers, but must begin with a letter. Avoid using the names of keywords and builtin functions as variable names; you will sometimes get odd behavior and it is really difficult to debug such errors.

We will want to use various constants, like the maximum number of iterations, convergence criterion, etc. We could hard-code these numbers into the loops and if statements, but this will be inconvenient to change. Better to define them at the front of the file as variables so we can configure them easily later on (i.e., if you want to have a maximum of 100 iterations, instead of writing `for ct in range(100)`in the code, define `maxiter = 100` at the top of the file and then use `for ct in range(maxiter)`).

3.4.3 Implementing logic

The main logic really involves looping up to the maximum number of iterations, performing the bisection and checking which half the root is in, and testing for convergence. We also have to figure out a way to evaluate the polynomial. Let's look at the test of whether the root is in the interval [xl, xtest]. We want to check whether $f(xl) \cdot f(xtest) < 0$. We could have a line like

```
if f(xl)* f(xtest)< 0
```

However, this requires that we evaluate `f(xl)` and `f(xtest)` on every loop, and then multiply them together. Since we must have evaluated one of the two on the previous loop, this is wasteful, as is the multiplication. Instead, you could do:

```
fxt = f(xtest)
if np.sign(fxl) != np.sign(fxt)
```

Using the builtin sign function in numpy. This will avoid one function call and one multiplication every time we go around the loop, making the program faster. In general, function evaluations are expensive to execute and you should try to minimize how many you make.

We also have to decide on convergence and which value we'll actually return. Our best estimate is the middle of the current interval – the next loop's `xtest`. This is the value we should return. To test convergence, we need to remember the previous value for comparison.

3.4.4 Testing

Now, let's check the algorithm's behavior. Try some polynomials that you know the root for (x-1 for example). You can also compare your results against suitable in-built numpy functions to check their validity .

3.5 Bisection method Tasks

3.5.1 Task 1 : Optional Bonus!

Submit the latest version of your flowchart for the Bisection method.

3.5.2 Task 2

Write a function `bisection(polyf, xl, xu)` that will accept an input polynomial function and a search interval, and return the root using the bisection method.

You should use your function mypolyval (from the last section) to evaluate the polynomial. Call this file `my_bisection.py`.

3.5.3 Task 3

Test your bisection function for a few polynomials (use Python's built-in `bisect` function for the "correct" values). Again this test could be done only when the file is run as a script but that's not mandatory.

4 Checklist of submission items

This is included for your convenience.

1. Flowcharts for Horner's scheme and Bisection methods (each an additional 5% grade if done correctly).
2. `mypolyval` function included in a file `scheme.py`.
3. `bisection` function in a file `my_bisection.py`.