HW 1

Jake Daniels A02307117

September 2022

Task 1

Define $f(x) = xe^{-x}$. Our first two guesses for our fixed point functions are:

$$g_1(x) = x - f(x)$$
$$g_2(x) = x + f(x)$$

Notice that when $f(x_0) = 0$, $g(x_0) = x_0$. Therefore any point that fixes $g(x)$ is a root to $f(x)$. Here is the code I wrote that given a function, initial guess, tolerance, and max amount of iterations, will return the approximation using both g_1 and g_2.

```
import numpy as np

def fn():
    f = input("Enter function you want to find the roots for (in terms of x): ")
    return f

def initialGuess():
    x0 = eval(input("Input initial guess for root here: "))
    return x0

def tolerance():
    tol = eval(input("Input tolerance here: "))
    return tol

def maxIter():
    mI = eval(input("Input max number of iterations here: "))
    return mI

def g1(f, x):
    gval = x - eval(f)
    return gval

def g2(f, x):
    gval = x + eval(f)
    return gval

def functionalIteration():
    f = fn()
    x0 = initialGuess()
    y0 = x0
    tol = tolerance()
    mI = maxIter()
    error = 10 * tol
    iter = 0
    while error > tol and iter < mI:
        x1 = g1(f, x0)
        error = np.abs(x1 - x0)
        x0 = x1
        iter = iter + 1
    error = 10 * tol
    iter = 0
    while error > tol and iter < mI:
        y1 = g2(f, y0)
        error = np.abs(y1 - y0)
        y0 = y1
        iter = iter + 1
    print("g1 approximates the root of f as x = ", x0)
    print("g2 approximates the root of f as x = ", y0)
```

Here is an attempt at finding the root $x = 0$. Note: the sentences in the following code are part of my user interface.

```
Enter function you want to find the roots for (in terms of x): x*np.exp(-x)
Input initial guess for root here: 2
Input tolerance here: .00000001
Input max number of iterations here: 25

g1 approximates the root of f as x =  9.860761315262648e-32
g2 approximates the root of f as x =  4.639207134292721
```

Here are some other guesses and approximations using the same function, tolerance, and max iterations:

```
Input initial guess for root here: -4

g1 approximates the root of f as x =  214.39260013257694
g2 approximates the root of f as x =  -inf

Input initial guess for root here: .5

g1 approximates the root of f as x =  4.676720855835697e-24
g2 approximates the root of f as x =  4.416480743133847

Input initial guess for root here: 19

g1 approximates the root of f as x =  18.99999733866849
g2 approximates the root of f as x =  19.0000026613251
```

Notice how both functions seem to approximate $x = 19$ as a fixed point. That is because our function, f, approaches 0 as x approaches infinity. So as x gets larger f will get very close to zero, but there will not be a root x^* such that $f(x^*) = 0$. Also both functions at $x = -4$ don't converge. This is because recall for convergence we need $|g'(x)| < 1$. If we analyze the derivatives we find:

$$g_1'(x) = x - (e^{-x} - xe^{-x}) \implies |g_1'(x)| < 1 \;\forall\; -0.37482\cdots < x < 1$$
$$g_2'(x) = x + (e^{-x} - xe^{-x}) \implies |g_2'(x)| < 1 \;\forall\; x > 1$$

So both values have derivatives greater than 1 at $x = -4$ which explains why they diverge as they should.

Task 2

The next step was to implement code to allow the user to get a table of guesses. I did this by implementing the following code. First I defined two new functions that would ask the user if they wanted a table and create a table. The function to ask the user if they want a table was implemented in the functionIteration() function as follows:

```
def wantTable():
    print("Do you want a table?")
    print("\t1) Yes")
    print("\t2) No")
    choice = eval(input("Enter here: "))
    return choice

def functionalIteration():
    f = fn()
    x0 = initialGuess()
    y0 = x0
    tol = tolerance()
    mI = maxIter()
    choice = wantTable()
    error = 10 * tol
    iter = 0
```

Next was creating the function that would actually make a table for the user. I created the following function, and used if else statements to create a table if the user selects that they want one:

```
def table(a,b,c):
    print("(1)" , a, " (2) ", b, " (3) ", c)

if choice == 1:
        print("Table for g1:")
        while error > tol and iter < mI:
            x1 = g1(f, x0)
            error = np.abs(x1 - x0)
            x0 = x1
            table(iter + 1, x0, error)
            iter = iter + 1
        iter = 0
        error = 10 * tol
        print("Table for g2:")
        while error > tol and iter < mI:
            y1 = g2(f, y0)
            error = np.abs(y1 - y0)
            y0 = y1
            table(iter + 1, y0, error)
            iter = iter + 1
    elif choice == 2:
        while error > tol and iter < mI:
            x1 = g1(f, x0)
            error = np.abs(x1 - x0)
            x0 = x1
            iter = iter + 1
        error = 10 * tol
        iter = 0
        while error > tol and iter < mI:
            y1 = g2(f, y0)
            error = np.abs(y1 - y0)
            y0 = y1
            iter = iter + 1
        print("g1 approximates the root of f as x = ", x0)
        print("g2 approximates the root of f as x = ", y0)
```

So if the user wants a table then the function runs as before, but now passes values for the iteration number, guess, and error through the table function which creates a table that looks like the following:

```
Table for g1:
(1) 1  (2)  1.7293294335267746  (3)  0.2706705664732254
(1) 2  (2)  1.4225406319355478  (3)  0.3067888015912268
(1) 3  (2)  1.0795651040295313  (3)  0.3429755279060165
(1) 4  (2)  0.7127901507893283  (3)  0.36677495324020304
(1) 5  (2)  0.36332742438777244  (3)  0.34946272640155585
(1) 6  (2)  0.11068453205380452  (3)  0.2526428923339679
(1) 7  (2)  0.011597401528274598  (3)  0.09908713052552992
(1) 8  (2)  0.00013372280487390469  (3)  0.011463678723400693
(1) 9  (2)  1.788059299516649e-08  (3)  0.00013370492428090952
(1) 10  (2)  3.1971560349601026e-16  (3)  1.7880592675450888e-08
(1) 11  (2)  9.860761315262648e-32  (3)  3.1971560349601016e-16
Table for g2:
(1) 1  (2)  2.2706705664732256  (3)  0.2706705664732256
(1) 2  (2)  2.5051012388781317  (3)  0.23443067240490612
(1) 3  (2)  2.7096861675800756  (3)  0.2045849287019439
(1) 4  (2)  2.890036623310162  (3)  0.18035045573008635
(1) 5  (2)  3.0506480309131114  (3)  0.16061140760294945
(1) 6  (2)  3.1950298461508506  (3)  0.1443818152377392
(1) 7  (2)  3.325915210937894  (3)  0.13088536478704338
(1) 8  (2)  3.44544731107422  (3)  0.11953210013632587
(1) 9  (2)  3.555324327464022  (3)  0.10987701638980196
(1) 10  (2)  3.656907433481829  (3)  0.10158310601780718
(1) 11  (2)  3.7513003159004574  (3)  0.09439288241862842
(1) 12  (2)  3.8394078012758794  (3)  0.08810748537542201
(1) 13  (2)  3.921979386268971  (3)  0.08257158499309147
(1) 14  (2)  3.9996418745543942  (3)  0.07766248828542333
(1) 15  (2)  4.072924110347538  (3)  0.07328223579314352
(1) 16  (2)  4.142275928589266  (3)  0.06935181824172787
(1) 17  (2)  4.208082830716638  (3)  0.06580690212737217
(1) 18  (2)  4.2706774679734805  (3)  0.06259463725684267
(1) 19  (2)  4.330348715193069  (3)  0.05967124721958861
(1) 20  (2)  4.387348907249922  (3)  0.05700019205685258
(1) 21  (2)  4.441899660621069  (3)  0.054550753371147565
(1) 22  (2)  4.494196595079639  (3)  0.0522969344585702
(1) 23  (2)  4.544413192744849  (3)  0.050216597665209584
(1) 24  (2)  4.592703974814426  (3)  0.04829078206957682
(1) 25  (2)  4.639207134292721  (3)  0.046503159478294975
```

Task 3

Now the goal is to apply functional iteration to find the roots of the following function:

$$f(x) = 10.14\, e^{x^2} \cos(\frac{\pi}{x})$$

Using the above g_1, g_2 if we want to find for what values of x, g converges, we need to find when the following functions are less than 1:

$$|g_1'(x)| = |1 - 10.14(2xe^{x^2}\cos(\frac{\pi}{x}) + \frac{\pi e^{x^2}}{x^2}\sin(\frac{\pi}{x}))|$$
$$|g_2'(x)| = |1 + 10.14(2xe^{x^2}\cos(\frac{\pi}{x}) + \frac{\pi e^{x^2}}{x^2}\sin(\frac{\pi}{x}))|$$

This is not going to be easy, and you are not going to get very large intervals for x when this is even true. This gave me the following idea. If we want to find the values of x that will be roots of our original function, we want to find for what values of x, $cos(\frac{\pi}{x}) = 0$. This is because $10.14\,e^{x^2} \neq 0 \;\forall\; x \in [\,-3, 7\,]$. So what if we change our functions for g to the following functions:

$$g_3(x) = x - \frac{1}{10.14\,e^{x^2}}f(x)$$
$$g_4(x) = x + \frac{1}{10.14\,e^{x^2}}f(x)$$

Since $\forall\; x \in [\,-3, 7\,]$ if $f(x^*) = 0$, $\frac{1}{10.14\,e^{(x^*)^2}}f(x^*) = 0$. Meaning g_3, g_4 will still be fixed at the roots of $f(x)$. Simplifying our $g's$ yields:

$$g_3(x) = x - \cos(\frac{\pi}{x})$$
$$g_4(x) = x + \cos(\frac{\pi}{x})$$

These are much easier functions to deal with, and will give us much bigger intervals of x to work with. However the derivatives still oscillate rapidly as $x \longrightarrow 0$, so the two most notable intervals are as follows:

$$|g_3'(x)| = |1 - \frac{\pi}{x^2}\sin(\frac{\pi}{x})| < 1 \;\forall\; x \in (\,-1, -.864\,) \cup (\,1, \infty\,)$$
$$|g_4'(x)| = |1 + \frac{\pi}{x^2}\sin(\frac{\pi}{x})| < 1 \;\forall\; x \in (\,-\infty, -1) \cup (\,.864, 1\,)$$

Using four values of x, namely $x = -2.5, -.9, .87, 5$ we get the following approximations (Note: g_1 is equivalent to g_3 in our case, same goes for 2 and 4):

```
Enter function you want to find the roots for (in terms of x): np.cos(np.pi / x)
Input initial guess for root here: -2.5
Input tolerance here: .00000001
Input max number of iterations here: 50

g1 approximates the root of f as x =  -49.26975153962745
g2 approximates the root of f as x =  -2.0000000005880794

Input initial guess for root here: -.9

g1 approximates the root of f as x =  1.9999999988603718
g2 approximates the root of f as x =  -1.9999999989974375

Input initial guess for root here: .87

g1 approximates the root of f as x =  1.9999999988577117
g2 approximates the root of f as x =  -1.9999999989084527

Input initial guess for root here: 5

g1 approximates the root of f as x =  2.000000001788166
g2 approximates the root of f as x =  53.934077312105074
```

So at all four of these values, either g_3 or g_4 approximated roots of $f(x)$ at $x^* = 2, -2$.

Task 4

Here is the code I wrote for the bisection method implementing everything including the functions for a table as well as a for loop ranging from 1 to whatever value k is based on the tolerance and interval.

```
import numpy as np

def fn():
    f = input("Enter function you want to find the roots for (in terms of x): ")
    return f

def f(fn, x):
    fval = eval(fn)
    return fval

def tolerance():
    tol = eval(input("Input tolerance here: "))
    return tol

def maxIter():
    mI = eval(input("Input max number of iterations here: "))
    return mI

def wantTable():
    print("Do you want a table?")
    print("\t1) Yes")
    print("\t2) No")
    choice = eval(input("Enter here: "))
    return choice

def table(a,b,c):
    print("(1)" , a, " (2) ", b, " (3) ", c)

def bisectMethod():
    fun = fn()
    a, b = eval(input("Enter interval for function as (a,b) where b>a: "))
    fa = f(fun, a)
    fb = f(fun, b)
    tol = tolerance()
    choice = wantTable()
    k = int(-np.log(tol / np.abs(b - a)) / np.log(2)) + 1
    if choice == 1:
        for i in range(1, k):
            error = np.abs(b-a)
            c = .5 * (a + b)
            fc = f(fun, c)
            if fa * fb > 0:
                print("Pick different a and b")
                print("f(a) x f(b) is positive")
                break
            elif fa * fc < 0:
                b = c
                fb = fc
            elif fb * fc < 0:
                a = c
                fa = fc
            elif fb * fa * fc == 0:
                if fa == fb == 0:
                    print("Both endpoints are roots no table could be formed.")
                    break
                elif fa == 0:
                    print("The left endpoint, ", a, ", is a root no table could be formed.")
                    break
                elif fb == 0:
                    print("The right endpoint, ", b, ", is a root no table could be formed.")
                    break
                elif fc == 0:
                    table(i, c, error)
                    break
            table(i, c, error)
    elif choice == 2:
        for i in range(1, k):
            c = .5 * (a + b)
            fc = f(fun, c)
            if fa * fb > 0:
                print("Pick different a and b")
                print("f(a) x f(b) is positive")
                c = " "
                break
            elif fa * fc < 0:
                b = c
                fb = fc
            elif fb * fc < 0:
                a = c
                fa = fc
            elif fb * fa * fc == 0:
                if fa == fb == 0:
                    print(a, "and", b, "are roots")
                    break
                elif fa == 0:
                    print(a, "is a root")
                    c = "given above"
                    break
                elif fb == 0:
                    print(b, "is a root")
                    c = "given above"
                    break
                elif fc == 0:
                    print(c, "is a root")
                    c = "given above"
                    break
        print("Root approximation: ", c)
```

Now to show it works, I can work on the two functions given above. For the function $f(x) = xe^{-x}$ we know the only root is at $x = 0$ so the bisection method should only work when we pick an interval with a negative left endpoint and a positive right endpoint. That can be seen below. Also assume tolerance is equal to .00000001 for all problems to save space.

```
Enter function you want to find the roots for (in terms of x): x*np.exp(-x)
Enter interval for function as (a,b) where b>a: -10, 14

(1) 1  (2)  2.0  (3)  24
(1) 2  (2)  -4.0  (3)  12.0
(1) 3  (2)  -1.0  (3)  6.0
(1) 4  (2)  0.5  (3)  3.0
(1) 5  (2)  -0.25  (3)  1.5
(1) 6  (2)  0.125  (3)  0.75
(1) 7  (2)  -0.0625  (3)  0.375
(1) 8  (2)  0.03125  (3)  0.1875
(1) 9  (2)  -0.015625  (3)  0.09375
(1) 10  (2)  0.0078125  (3)  0.046875
(1) 11  (2)  -0.00390625  (3)  0.0234375
(1) 12  (2)  0.001953125  (3)  0.01171875
(1) 13  (2)  -0.0009765625  (3)  0.005859375
(1) 14  (2)  0.00048828125  (3)  0.0029296875
(1) 15  (2)  -0.000244140625  (3)  0.00146484375
(1) 16  (2)  0.0001220703125  (3)  0.000732421875
(1) 17  (2)  -6.103515625e-05  (3)  0.0003662109375
(1) 18  (2)  3.0517578125e-05  (3)  0.00018310546875
(1) 19  (2)  -1.52587890625e-05  (3)  9.1552734375e-05
(1) 20  (2)  7.62939453125e-06  (3)  4.57763671875e-05
(1) 21  (2)  -3.814697265625e-06  (3)  2.288818359375e-05
(1) 22  (2)  1.9073486328125e-06  (3)  1.1444091796875e-05
(1) 23  (2)  -9.5367431640625e-07  (3)  5.7220458984375e-06
(1) 24  (2)  4.76837158203125e-07  (3)  2.86102294921875e-06
(1) 25  (2)  -2.384185791015625e-07  (3)  1.430511474609375e-06
(1) 26  (2)  1.1920928955078125e-07  (3)  7.152557373046875e-07
(1) 27  (2)  -5.960464477539063e-08  (3)  3.5762786865234375e-07
(1) 28  (2)  2.9802322387695312e-08  (3)  1.7881393432617188e-07
(1) 29  (2)  -1.4901161193847656e-08  (3)  8.940696716308594e-08
(1) 30  (2)  7.450580596923828e-09  (3)  4.470348358154297e-08
(1) 31  (2)  -3.725290298461914e-09  (3)  2.2351741790771484e-08

or

Root approximation:  -3.725290298461914e-09
```

Notice above how the error terms are decreasing by $\frac{1}{2}$ which is correct since the error is the size of the interval which is being halved each iteration. Here is an example when the both $f(a)$ and $f(b)$ are positive, so our intermediate value theorem does not guarantee a root so we cannot use the bisection method. This works the same for when they're both negative. So applying the same for our first function, we know there are roots at $x = 2, -2$, so if we pick intervals around those points we better get the same conclusion.

```
Enter function you want to find the roots for (in terms of x): 10.14*np.exp(x*x)*np.cos(np.pi/x)
Enter interval for function as (a,b) where b>a: -2.1,-1.6
Input tolerance here: .00000001

Root approximation:  -1.9999999910593034

Enter interval for function as (a,b) where b>a: 1.5, 2.6

(1) 1  (2)  2.05  (3)  1.1
(1) 2  (2)  1.775  (3)  0.5499999999999998
(1) 3  (2)  1.9124999999999999  (3)  0.2749999999999999
(1) 4  (2)  1.9812499999999997  (3)  0.13749999999999996
(1) 5  (2)  2.015625  (3)  0.06875000000000009
(1) 6  (2)  1.9984374999999999  (3)  0.034375000000000266
(1) 7  (2)  2.00703125  (3)  0.017187500000000133
(1) 8  (2)  2.0027343749999997  (3)  0.008593749999999956
(1) 9  (2)  2.0005859375  (3)  0.004296874999999867
(1) 10  (2)  1.99951171875  (3)  0.0021484375000000444
(1) 11  (2)  2.000048828125  (3)  0.0010742187499999112
(1) 12  (2)  1.9997802734375  (3)  0.0005371093750001776
(1) 13  (2)  1.9999145507812501  (3)  0.0002685546875000888
(1) 14  (2)  1.9999816894531253  (3)  0.0001342773437500444
(1) 15  (2)  2.0000152587890625  (3)  6.713867187491118e-05
(1) 16  (2)  1.9999984741210939  (3)  3.3569335937233546e-05
(1) 17  (2)  2.0000068664550783  (3)  1.6784667968616773e-05
(1) 18  (2)  2.000002670288086  (3)  8.392333984419409e-06
(1) 19  (2)  2.00000057220459  (3)  4.196166992320727e-06
(1) 20  (2)  1.9999995231628418  (3)  2.098083496049341e-06
(1) 21  (2)  2.0000000476837156  (3)  1.0490417481356928e-06
(1) 22  (2)  1.9999997854232787  (3)  5.245208738458018e-07
(1) 23  (2)  1.9999999165534972  (3)  2.622604369229009e-07
(1) 24  (2)  1.9999999821186063  (3)  1.3113021846145045e-07
(1) 25  (2)  2.000000014901161  (3)  6.556510934174753e-08
(1) 26  (2)  1.9999999985098837  (3)  3.278255489291837e-08
```

As it can be seen above, we were able to approximate the same roots as when we used functional iteration. We can even approximate a few more roots as follows:

```
Enter interval for function as (a,b) where b>a: -1, .5

Root approximation:  -0.6666666753590107

Enter interval for function as (a,b) where b>a: .3,.5

Root approximation:  0.39999998807907106
```

Note these roots are approximately $\frac{2}{3}$ and $\frac{2}{5}$.

Task 5

A folder for the root finding code above has been created in github and has been downloaded onto my computer. A file has been created in it called README to give a synopsis of what the repository is for. Below is the link to said repository.

https://jake-daniels16.github.io/math4610/