Math 4610 – HW 2

Jake Daniels A02307117

September 2022

Task 1:

Before I start, here is the link to my repository and software manual.

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

There you can find the pages of my software manual that were asked for in their respected tasks. Additionally, all software manuals are written with the changes made by allowing the user the option of a table to show their results.
Here is the code for Newton's Method:

```
import numpy as np

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

def fPrime():
    fp = input("Enter derivative here: ")
    return fp

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

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 newtonsMethod():
    fun = fn()
    fp = fPrime()
    x0 = initialGuess()
    tol = tolerance()
    mI = maxIter()
    error = 10 * tol
    iter = 0
    while error > tol and iter < mI:
        f0 = f(fun, x0)
        fp0 = f(fp, x0)
        x1 = x0 - f0 / fp0
        error = np.abs(x1 - x0)
        iter = iter + 1
        x0 = x1
    print("Root approximation: ", x0)
```

Now, to show the code works, I will use the function, $f(x) = xe^{-x}$, as an example to try and find it's root at $x = 0$:

```
Enter function you want to find the roots for (in terms of x): x*np.exp(-x)
Enter derivative here: np.exp(-x) - x*np.exp(-x)
Input initial guess for root here: .5
Input tolerance here: .00000001
Input max number of iterations here: 25
```

We need to input the functions as such using the numpy module and construction of python. The following input will yield the following output:

```
Root approximation:  -8.80999858950826e-27
```

This is approximately zero so the code works correctly and was able to attain the root.

Task 2:

Here is my code for the secant method:

```
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 initialGuesses():
    x0, x1 = eval(input("Input initial guesses for root here as (x0, x1): "))
    return x0, x1

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

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

def secantMethod():
    fun = fn()
    x0, x1 = initialGuesses()
    tol = tolerance()
    mI = maxIter()
    error = 10 * tol
    iter = 0
    f0 = f(fun, x0)
    f1 = f(fun, x1)
    while error > tol and iter < mI:
        x2 = x1 - (f1 * (x1 - x0)) / (f1 - f0)
        error = np.abs(x2 - x1)
        iter = iter + 1
        x0 = x1
        x1 = x2
        f0 = f1
        f1 = f(fun, x2)
    print("Root approximation: ", x1)
```

To show the code works, I will use the function, $f(x) = xe^{-x}$, as an example to try and find it's root at $x = 0$:

```
Enter function you want to find the roots for (in terms of x): x*np.exp(-x)
Input initial guesses for root here as (x0, x1): -1, 1
Input tolerance here: .00000001
Input max number of iterations here: 25
```

We need to input the functions as such using the numpy module and construction of python. The following input will yield the following output:

```
Root approximation:  1.549984207858443e-15
```

Which is approximately zero, so using this approximation we can guess that the function above has a root at or near $x = 0$. Which is what we know is true, so the code works.

Task 3:

To create a table when wanted, I created the following two functions to ask the user if they want a table and create the table if so.

```
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)
```

Newton's Method was changed by adding in this choice parameter and changing the loop to the following:

```
if choice == 1:
        while error > tol and iter < mI:
            f0 = f(fun, x0)
            fp0 = f(fp, x0)
            x1 = x0 - f0 / fp0
            error = np.abs(x1 - x0)
            table(iter + 1, x1, error)
            iter = iter + 1
            x0 = x1
elif choice == 2:
    while error > tol and iter < mI:
        f0 = f(fun, x0)
        fp0 = f(fp, x0)
        x1 = x0 - f0 / fp0
        error = np.abs(x1 - x0)
        iter = iter + 1
        x0 = x1
    print("Root approximation: ", x0)
```

This was implemented in a similar manner for the secant method as follows:

```
if choice == 1:
        while error > tol and iter < mI:
            x2 = x1 - (f1 * (x1 - x0)) / (f1 - f0)
            error = np.abs(x2 - x1)
            table(iter + 1, x2, error)
            iter = iter + 1
            x0 = x1
            x1 = x2
            f0 = f1
            f1 = f(fun, x2)
    elif choice == 2:
        while error > tol and iter < mI:
            x2 = x1 - (f1 * (x1 - x0)) / (f1 - f0)
            error = np.abs(x2 - x1)
            iter = iter + 1
            x0 = x1
```

```
            x1 = x2
            f0 = f1
            f1 = f(fun, x2)
        print("Root approximation: ", x1)
```

Using the same example from Task 1, we get the following table for Newton's method with the included changes to create a table:

```
(1) 1  (2)  -0.5  (3)  1.0
(1) 2  (2)  -0.16666666666666669  (3)  0.3333333333333333
(1) 3  (2)  -0.023809523809523808  (3)  0.14285714285714288
(1) 4  (2)  -0.0005537098560354364  (3)  0.023255813953488372
(1) 5  (2)  -3.0642493416461764e-07  (3)  0.0005534034311012718
(1) 6  (2)  -9.389621148813321e-14  (3)  3.0642484026840615e-07
(1) 7  (2)  -8.80999858950826e-27  (3)  9.38962114881244e-14
```

Using the same example as above for the secant method, we get the following table:

```
(1) 1  (2)  0.7615941559557649  (3)  0.23840584404423515
(1) 2  (2)  -6.145115192053641  (3)  6.9067093480094055
(1) 3  (2)  0.7607373842425833  (3)  6.905852576296224
(1) 4  (2)  0.7598809490516972  (3)  0.0008564351908860734
(1) 5  (2)  -2.4117305197619667  (3)  3.171611468813664
(1) 6  (2)  0.7185207502140756  (3)  3.1302512699760423
(1) 7  (2)  0.6782842464737974  (3)  0.04023650374027821
(1) 8  (2)  -1.6152103158907716  (3)  2.293494562364569
(1) 9  (2)  0.5850439821307694  (3)  2.200254298021541
(1) 10  (2)  0.5001671785336054  (3)  0.08487680359716399
(1) 11  (2)  -0.6389152674058604  (3)  1.1390824459394657
(1) 12  (2)  0.2719161660239241  (3)  0.9108314334297845
(1) 13  (2)  0.13879672313573155  (3)  0.13311944288819255
(1) 14  (2)  -0.04740627021736338  (3)  0.18620299335309493
(1) 15  (2)  0.00687409861557263  (3)  0.05428036883293601
(1) 16  (2)  0.0003193254984228877  (3)  0.006554773117149742
(1) 17  (2)  -2.202990601857084e-06  (3)  0.0003215284890247476
(1) 18  (2)  7.035826268411442e-10  (3)  2.203694184483925e-06
(1) 19  (2)  1.549984207858443e-15  (3)  7.035810768569363e-10
```

Notice that the last entry of (2) in both cases was the exact same root given above. So the code for the table works well and does not mess up the code.

Task 4:

Here is the code for the newton hybrid method:

```
import numpy as np

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

def fPrime():
    fp = input("Enter derivative here: ")
    return fp

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 newtonHybrid():
    fun = fn()
    fp = fPrime()
    a, b = eval(input("Enter interval for function as (a,b) where b > a: "))
    tol = tolerance()
    mI = maxIter()
    # choice = wantTable()
    x0 = .5 * (a + b)
    error = 10 * tol
    iter = 0
    while error > tol and iter < mI:
        f0 = f(fun, x0)
        fp0 = f(fp, x0)
        x1 = x0 - f0 / fp0
        newtError = np.abs(x1 - x0)
        if newtError > error:
            fa = f(fun, a)
            fb = f(fun, b)
            for i in range(1, 4):
                c = .5 * (a + b)
                fc = f(fun, c)
                if fa * fc < 0:
                    b = c
                    fb = fc
                elif fb * fc < 0:
                    a = c
                    fa = fc
            error = np.abs(b - a)
            x0 = .5 * (a + b)
        else:
            x0 = x1
            error = newtError
            iter = iter + 1
    print("Root approximation: ", x0)
```

To show my code works for the hybrid Newton Bisection method, we will look at the function, $f(x) = 10.14e^{x^2}cos(\frac{\pi}{x})$ and try and find a root on the interval $[-3, 7]$.

```
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 derivative here: 10.14 * (2 * x * np.exp(x * x) * np.cos(np.pi / x) + (np.pi /  (x * x)) * np.exp(x * x) * np.sin(np.pi /
↪  x))
Enter interval for function as (a,b) where b > a: -1.468, 5.43
Input tolerance here: .00000001
Input max number of iterations here: 25
```

These inputs yield the following approximation:

```
Root approximation:  1.9999999999999998
```

This is approximately 2, if we plug $x = 2$ into the function you will have a constant multiplied by $cos(\frac{\pi}{2})$ which is zero, so $f(2) = 0$ and our code works correctly! As a second example, here it is working again to find the root $x = \frac{2}{3}$:

```
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 derivative here: 10.14 * (2 * x * np.exp(x * x) * np.cos(np.pi / x) + (np.pi /  (x * x)) * np.exp(x * x) * np.sin(np.pi /
↪  x))
Enter interval for function as (a,b) where b > a: .5, 1.4
Input tolerance here: .00000001
Input max number of iterations here: 25
Root approximation:  0.6666666666666667
```

Task 5

Here is the code for my secant bisect hybrid:

```
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 secantHybrid():
    fun = fn()
    a, b = eval(input("Enter interval for function as (a,b) where b > a: "))
    tol = tolerance()
    mI = maxIter()
    x0, x1 = a, b
    error = 10 * tol
    iter = 0
    f0 = f(fun, x0)
    f1 = f(fun, x1)
    while error > tol and iter < mI:
        x2 = x1 - ((f1 * (x1 - x0)) / (f1 - f0))
        secError = np.abs(x2 - x1)
        if secError > error:
            a = x1
            b = x2
            fa = f0
            fb = f1
            for i in range(1, 4):
                c = .5 * (a + b)
                fc = f(fun, c)
                if fa * fc < 0:
                    b = c
                    fb = fc
                elif fb * fc < 0:
                    a = c
                    fa = fc
                else:
                    break
            error = np.abs(b - a)
            iter = iter + 1
            x0, x1 = a, b
            f0 = fa
            f1 = fb
        else:
            x0 = x1
            x1 = x2
            f0 = f1
            f1 = f(fun, x2)
            error = secError
            iter = iter + 1
    print("Root approximation: ", x1)
```

To show my code works for the hybrid Secant Bisection method, we will look at the function, $f(x) = 10.14e^{x^2}cos(\frac{\pi}{x})$ and try and find a root on the interval $[-3, 7]$.

```
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: .694, 3.521
Input tolerance here: .00000001
Input max number of iterations here: 25
Root approximation:  2.0000000019697475
```

The approximation is approximately 2, which as we stated above is a root. Here is one more example of the code working correctly.

```
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: .5, 1
Input tolerance here: .00000001
Input max number of iterations here: 25
Root approximation:  0.6666666679084301
```

The approximation is approximately $\frac{2}{3}$ which again as stated above, is a root.