University of Colorado
Boulder

# HW4

Jesse Hettleman
APPM 4600

College of Engineering and Applied Science
Department of Applied Math

September 27, 2024

# 1 Problem 1

### 1.a

In this problem, we are given the equation for temperature $x$ meters below the surface after $t$ seconds:

$$\frac{T(x,t) - T_s}{T_i - T_s} = erf(\frac{x}{2\sqrt{\alpha t}})$$

where $T_i$ is the initial soil temperature, $T_s$ is the constant temperature during the cold period, and $\alpha$ is the thermal conductivity constant.

We want to find the depth, $x$, to bury a pipe so that it will only freeze (i.e. $T(x,t) = 0$) after 60 days $= 5,184,000$ seconds given $T_i = 20$, $T_s = -15$, and $\alpha = 0.138 * 10^{-6}$.

We will begin by manipulating the given equation to frame this question as a root finding problem:

$$\frac{T(x,t) - T_s}{T_i - T_s} = erf(\frac{x}{2\sqrt{\alpha t}}) \quad (1)$$

$$\frac{0-(-15)}{20-(-15)} = erf(\frac{x}{2\sqrt{0.138 * 10^{-6} * 5.184 * 10^{6}}}) \quad (2)$$

$$0 = erf(\frac{x}{1.69161697792}) - \frac{3}{7} \quad (3)$$

Thus, in this root finding problem, $f(x) = erf(\frac{x}{1.69161697792}) - \frac{3}{7}$. Since $erf(x) = \frac{2}{\pi}\int_0^x e^{-t^2}dt$:

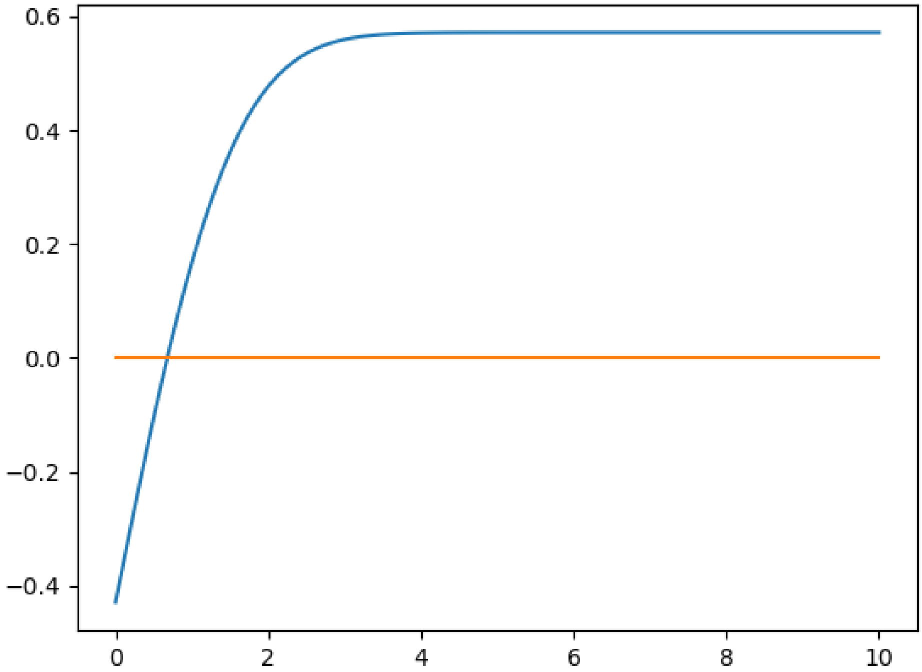
$$\frac{d}{dx}erf(x) = \frac{d}{dx}\frac{2}{\pi}\int_0^x e^{-t^2}dt \quad (4)$$

$$= \frac{2}{\pi}e^{-x^2} \quad (5)$$

Therefore, $f'(x) = \frac{2}{1.69161697792\pi}e^{-(\frac{x}{1.69161697792})^2}$.

Below is a plot of $f(x) = erf(\frac{x}{1.69161697792}) - \frac{3}{7}$ created using python code, found in 6.

**Figure 1: Plot of $f(x) > 0$**




### 1.b

Using the bisection method with $a = 0$ and $b = 2$, I calculate the approximate depth to be 0.676961854480453 meters.

### 1.c

Using the Newton-Raphson method with $x_0 = 0.01$, I calculate the approximate depth to be 0.6769617952849367 meters. This is equivalent to using the fixed point iteration where $g(x) = x - \frac{f(x)}{f'(x)}$. Code for this calculation and all others in this problem can be found in 6.

If I use the Newton-Raphson method with $x_0 = x_{hat}$ such that $f(x_{hat}) > 0$, then the root is still found and the method converges. When I pick $x_{hat} = 2$, the Newton method converges to $r = 0.6769618783825369$ which is slightly different from before after the 6th digit.

## 2 Problem 2

### 2.a

Mathematically, for a root $r$ to be of multiplicity $m$, there exists $h(x)$ such that $h(r) \neq 0$, then $f(r) = (x-r)^m h(r)$. This means $f(r) = f'(r) = ... = f^{m-1}(r) = 0$ and $f^m(r) \neq 0$.

### 2.b

We can equate Newton's method to a fixed point iteration with $g(x) = x - \frac{f(x)}{f'(x)}$ which is derived from the point slope form of finding zeros in Newton's method.

In this case, simplifying $g(x)$ when $f(x) = (x-r)^m h(x)$ and taking the limit of $g'(x)$ as $x \to r$ results in the following:

$$
\begin{aligned}
g(x) &= x - \frac{(x-r)^m h(x)}{m(x-r)^{m-1}h(x) + (x-r)^m h'(x)} && (6)\\
&= x - \frac{(x-r)h(x)}{mh(x) + (x-r)h'(x)} && (7)
\end{aligned}
$$

Furthermore, when we take the derivative of $g(x)$ using the product rule, and for the purpose of simplification denote all expressions with $(...)$ that go to 0 as $x \to r$ due to $(x-r) \to 0$:

$$
\begin{aligned}
g'(x) &= 1 - \frac{[h(x) + (x-r)(...)][mh(x) + (...)] - [(x-r)h(x)](...)}{mh(x) + (x-r)h'(x)} && (8)\\
\lim_{x \to r} g'(x) &= 1 - \frac{mh^2(x)}{m^2h^2(x)} && (9)\\
&= 1 - \frac{1}{m} && (10)
\end{aligned}
$$

Thus, if $m > 1$ then $g'(x) \in [\frac{1}{2}, 1)$ so the Newton method will not converge quadratically. Rather, the Newton method will converge linearly with rate $1 - \frac{1}{m}$.

**2.c**

However, if we apply the Newton method to $g(x) = x - m\frac{f(x)}{f'(x)}$, then following the calculation above we see:

$$
\begin{aligned}
g'(x) &= 1 - m\frac{[h(x) + (x-r)(...)][mh(x) + (...)] - [(x-r)h(x)](...)}{mh(x) + (x-r)h'(x)} && (11)\\
\lim_{x \to r} g'(x) &= 1 - m\frac{mh^2(x)}{m^2h^2(x)} && (12)\\
&= 1 - m\frac{1}{m} && (13)\\
&= 1 - 1 && (14)\\
&= 0 && (15)
\end{aligned}
$$

Therefore, in this case we achieve quadratic convergence which is the entire goal of the Newton method in the first place.

**2.d**

The manipulated expression for $g(x)$ in part c above provides a fix for achieving quadratic convergence when dealing with roots with multiplicity $m > 1$. This is actually a method for fixing Newton's method that we learned in lecture, where we set a new $g(x) = x - \alpha\frac{f(x)}{f'(x)}$. This this case, we set $\alpha = m$ so that $g'(r) = 0$.

## 3 Problem 3

We will begin this problem with the formula for order of convergence of a sequence $x_{k}{}_{k=1}^{\infty}$ that converges to $\alpha$ with order $p$ and manipulating to find logarithmic relationships:

$$\frac{|x_{k+1}-\alpha|}{|x_k-\alpha|^p} = C \tag{16}$$
$$|x_{k+1}-\alpha| = C|x_k-\alpha|^p \tag{17}$$
$$log(|x_{k+1}-\alpha|) = log(C|x_k-\alpha|^p) \tag{18}$$
$$log(|x_{k+1}-\alpha|) = p*log(|x_k-\alpha|)+log(C) \tag{19}$$

As exemplified within this logarithmic relationship, the order of convergence $p$ roughly determines how many valid digits you can get per iteration of the fixed point method. For example, if $p=1$, then you would get linear convergence, so you could expect one digit of accuracy per three or four iterations ($ln(3)=1.09$). Similarly, if $p=2$, then you could expect the digits of accuracy to *double* (not just two more accurate digits) every three or four iterations. Thus, $p$ clearly highlights the rate at which you will gain digits of accuracy when the order of convergence is displayed in this logarithmic manner.

## 4 Problem 4

In this problem, we will explore fixes to Newton's Method when finding a particular root of $f(x) = e^{3x} - 27x^6 + 27x^4e^x - 9x^2e^{2x}$ on the interval $(3,5)$.

First we will apply the regular Newton's Method, which is the equivalent of the Fixed Point iteration using $g(x) = x - \frac{f(x)}{f'(x)}$. To do so, we must find $f'(x)$:

$$f'(x) = 3e^{3x} - 162x^5 + 108x^3e^x + 27x^4e^x - 18xe^{2x} - 18x^2e^{2x}$$

Finding the root using Newton's Method, an initial guess of $x_0 = 4$, and error tolerance $tol = 10^{-10}$ yields root $r = 3.7330885607957707$. However, this took 46 iterations. It is important to see how many iterations the regular Newton Method took so that we can compare it to some fixes.

The first fix we will attempt is listed in problem 2.c: we set $g(x) = x - m\frac{f(x)}{f'(x)}$ where $m$ is the multiplicity of the root. I determined the multiplicity of the root between $(3,5)$ to be $m=3$ by looking at a graph of $f(x)$ and also by testing this method for both $m=3$ and $m=5$.

Finding the root using this fix to Newton's method with $m=3$ and $x_0=4$ yields root $r = 3.7330794366049385$ within 6 iterations! This is a significant improvement from the 46 iterations that the unaltered Newton's Method took to converge. Moreover, this method was relatively easy because no difficult derivatives were required.

Another fix we learned in class is to apply Newton's method to $f_{new} = \frac{f(x)}{f'(x)}$. This involves finding $f'_{new}(x)$ since:

$$g_{new}(x) = x - \frac{\frac{f(x)}{f'(x)}}{(\frac{f(x)}{f'(x)})'}$$

I found this derivative, and it is indeed extraordinarily long. Instead of type-setting this function into latex, I will simply copy the code version I already typed into python:

$(\frac{f(x)}{f'(x)})' =$ ((4374*(x**10)) + (729*(math.e**x)*(x**10)) - (2916*(math.e**x)*(x**9)) - (4374*(math.e**x)*(x**8)) - (972*(math.e**(2*x))*(x**8)) + (3888*(math.e**(2*x))*(x**7)) + (972*(math.e**(2*x))*(x**6)) + (486*(math.e**(3*x))*(x**6)) - (1944*(math.e**(3*x))*(x**5)) - (108*(math.e**(4*x))*(x**4)) + (324*(math.e**(3*x))*(x**4)) + (432*(math.e**(4*x))*(x**3)) + (9*(math.e**(5*x))*(x**2)) - (162**(math.e**(4*x))*(x**2)) - (36*(math.e**(5*x))) + (18*(math.e**(5*x)))) / ((f'(x))**2)

This function is so long, in fact, that when I ran Newton's Method using $g_{new}(x) = x - \frac{\frac{f(x)}{f'(x)}}{(\frac{f(x)}{f'(x)})'}$, python returned an error because the function was too large. Thus, this fix to Newton's Method did not work in this instance, and made the problem unsolvable numerically.

To summarize findings from this problem, the fix to Newton's Method involving multiplying $\frac{f(x)}{f'(x)}$ by $m$ is the preferable fix in this instance. Not only did this method converge to a root with only 6 iterations, but it also did not involve a miserably tedious quotient rule calculation yielding a 16-term polynomial.

# 5 Problem 5

## 5.a

I calculated the largest root of $f(x) = x^6 - x - 1$ using both the Newton-Raphson ("Newton") method and Secant method. Using an error tolerance of $tol = 1 * 10 - 10$, the Newton method yielded $r = 1.1347241384015194$ in 8 iterations, while the Secant method resulted in $r = 1.1347241384015194$ in 9 iterations.

The following tables display tables showing the errors and estimates for each iteration of the Newton and Secant methods for this problem respectively:

**Figure 2: Table of Errors for Newton Method**

Newton Method Error Table:

| | Iteration | Estimate | Error |
|---|---|---|---|
| 0 | 1 | [1.6806282722513088] | [0.5459041338497894] |
| 1 | 2 | [1.4307389882390624] | [0.296014849837543] |
| 2 | 3 | [1.2549709561094364] | [0.12024681770791701] |
| 3 | 4 | [1.1615384327733131] | [0.026814294371793723] |
| 4 | 5 | [1.1363532741705054] | [0.0016291357689859343] |
| 5 | 6 | [1.134730528343629] | [6.389942109663593e-06] |
| 6 | 7 | [1.1347241385002211] | [9.870171346904044e-11] |

For the Newton method, we expect quadratic convergence. This is what we notice occuring within this table, as the error is approximatly reduced by a factor of $\frac{1}{2}$ each iteration.

**Figure 3: Table of Errors for Secant Method**

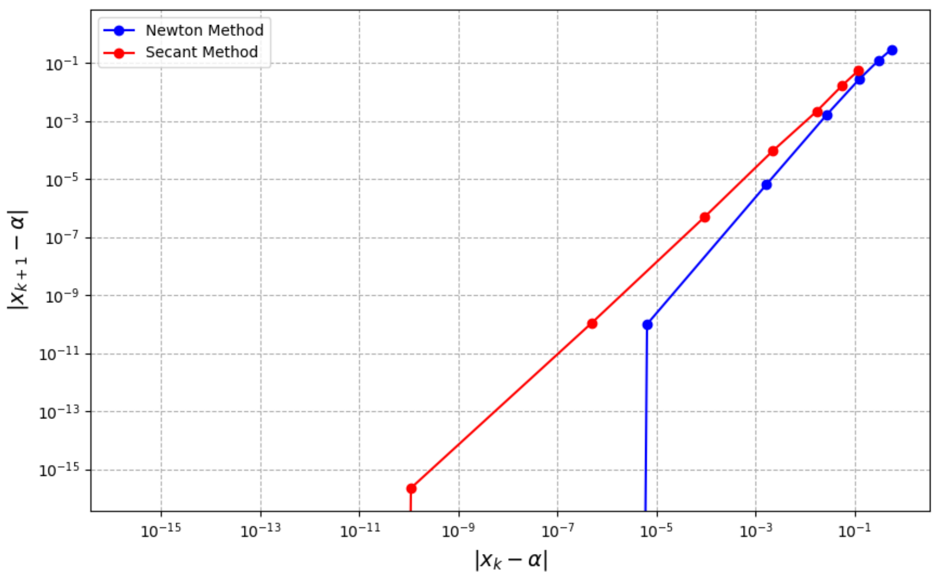
```
Secant Method Error Table:

   Iteration                 Estimate                   Error
0          1   [1.0161290322580645]      [0.5459041338497894]
1          2   [1.1905777686766374]       [0.296014849837543]
2          3   [1.1176558309415516]     [0.12024681770791701]
3          4    [1.132531550216133]    [0.026814294371793723]
4          5   [1.1348168080048529]   [0.0016291357689859343]
5          6    [1.134723645948705]   [6.389942109663593e-06]
6          7   [1.1347241382912159]   [9.870171346904044e-11]
7          8   [1.1347241384015196]                    [0.0]
```

For the Secant method, we expect between linear and quadratic convergence. In fact, we expect the order of convergence to be approximately equal to the golden ratio $\phi = 1.618$. Again, we observe what we expect. The Secant method does not converge as quickly as the Newton method for this function, which is supported by the fact that it requires one more iteration to achieve the desired level of tolerance.

NOTE: the reason all iterations do not appear in this chart is due to the fact that the first iterations has no $x_{n-1}$ to compare to. Thus, only the total number of iterations $-1$ appear in this table, and the subsequent chart.

**5.b**

**Figure 4: Log-Log Plot of $e_n$ vs $e_{n+1}$**




In this log-log plot, the slope of each line represents the order of convergence. The Newton method slope is slightly steeper, which is in line with the quadratic convergence we expect, whereas the Secant method has a slightly less steep slope, closer to the golden ratio $\phi = 1.618$. This interpretation of the slope as the order of convergence makes sense, especially when we recall Problem 3 (3) where we saw $log(|x_{k+1} - \alpha|) = p * log(|x_k - \alpha|) + log(C)$. This equation is evidently in $y = mx + b$ form when y is $log(|x_{k+1} - \alpha|)$, x is $log(|x_k - \alpha|)$, and $p$, the order of convergence, is the slope.

# 6 Appendix

## 6.1 Code

**Description of Code Here:**

```
import numpy as np
import math
import matplotlib.pyplot as plt
import scipy.special
import pandas as pd

def bisection(f,a,b,tol):

#     Inputs:
#      f,a,b       - function and endpoints of initial interval
#       tol  - bisection stops when interval length < tol

#     Returns:
#       astar - approximation of root
#       ier   - error message
#             - ier = 1 => Failed
#             - ier = 0 == success

#      first verify there is a root we can find in the interval
    count = 0

    fa = f(a)
    fb = f(b)
    if (fa*fb>0):
       ier = 1
       astar = a
       return [astar, ier, count]

#    verify end points are not a root
    if (fa == 0):
      astar = a
      ier =0
      return [astar, ier, count]

    if (fb ==0):
      astar = b
      ier = 0
      return [astar, ier, count]

    d = 0.5*(a+b)
    while (abs(d-a)> tol):
```

```
      fd = f(d)
      if (fd ==0):
        astar = d
        ier = 0
        return [astar, ier, count]
      if (fa*fd<0):
         b = d
      else:
        a = d
        fa = fd
      d = 0.5*(a+b)
      count = count +1
#       print('abs(d-a) = ', abs(d-a))

    astar = d
    ier = 0
    return [astar, ier, count]

def fixedpt(f,x0,tol,Nmax):

    ''' x0 = initial guess'''
    ''' Nmax = max number of iterations'''
    ''' tol = stopping tolerance'''

    approx = np.zeros((Nmax,1))

    count = 0
    while (count <Nmax):
       x1 = f(x0)
       approx[count] = x1
       if (abs(x1-x0) <tol):
          xstar = x1
          ier = 0
          count += 1
          return [xstar,ier,approx,count]
       x0 = x1
       count = count +1

    xstar = x1
    ier = 1
    return [xstar, ier,approx,count]

def secant(x0,x1,f,Nmax,tol):

  approx = np.zeros((Nmax,1))

  count = 0
  x2 = x1

  if abs(f(x1) - f(x0)) == 0:
    ier = 1
    xstar = x1
    approx[count] = x1
    return [xstar, ier,approx,count]

  for i in range(1,Nmax+1):
    x2 = x1 - (f(x1) * ((x1 - x0)/(f(x1) - f(x0))))
    approx[count] = x2
```

```
    count += 1

    if abs(x2 - x1) < tol:
      xstar = x2
      ier = 0
      return [xstar, ier,approx,count]

    x0 = x1
    x1 = x2

    if(abs(f(x1) - f(x0))) == 0:
      ier = 1
      xstar = x2
      return [xstar, ier,approx,count]

  xstar = x2
  ier = 1
  return [xstar, ier,approx,count]

# Question 1

def question1():
    print("-----Question 1------")

    f = lambda x: scipy.special.erf(x/1.69161697792) - (3/7)

    x = np.linspace(0,2,100)
    y = np.array([f(X) for X in x])

    plt.plot(x,y)
    plt.plot(x, [0 for X in x])
    plt.savefig("HW4.1.a.png")
    plt.clf()

    # Bisection
    print("Bisection Method:")

    a = 0
    b = 2

    tol = 1e-13

    [astar,ier,count] = bisection(f,a,b,tol)
    print('the approximate root is',astar)
    print('the error message reads:',ier)
    print('f(astar) =', f(astar))
    print('number of iterations = ', count)

    print("Newton Method:")

    f_deriv = lambda x: ...
        (2/(1.69161697792*math.pi))*math.e**((-x/1.69161697792)**2)

    g = lambda x: x - (f(x)/f_deriv(x))
    Nmax = 100
    tol = 1e-6

    x0 = 0.01
```

```
        [xstar,ier,approx,count] = fixedpt(g,x0,tol,Nmax)
        print('the approximate fixed point is:',xstar)
        print('g(xstar):',g(xstar))
        print('Error message reads:',ier)
        print(approx)

# question1()

# Question 4

def question4():

    print("Newton Method:")

    f = lambda x: (math.e**(3*x)) - (27*(x**6)) + ...
        (27*(x**4)*(math.e**x)) - (9*(x**2)*(math.e**(2*x)))
    f_deriv = lambda x: (3*math.e**(3*x)) - (162*(x**5)) + ...
        (108*(x**3)*(math.e**x)) + (27*(x**4)*(math.e**x)) - ...
        (18*(x)*(math.e**(2*x))) - (18*(x**2)*(math.e**(2*x)))

    g = lambda x: x - (f(x)/f_deriv(x))
    Nmax = 100
    tol = 1e-10

    x0 = 4
    [xstar,ier,approx,count] = fixedpt(g,x0,tol,Nmax)
    print('the approximate fixed point is:',xstar)
    print('g(xstar):',g(xstar))
    print('Error message reads:',ier)
    print('number of iterations = ', count)

    print("2c Fix -- Multiply By m Method:")

    m = 3
    g2 = lambda x: x - m*(f(x)/f_deriv(x))
    Nmax = 100
    tol = 1e-10

    x0 = 4
    [xstar,ier,approx,count] = fixedpt(g2,x0,tol,Nmax)
    print('the approximate fixed point is:',xstar)
    print('g(xstar):',g2(xstar))
    print('Error message reads:',ier)
    print('number of iterations = ', count)

    print("g = f/f' Fix Method:")

    f_divf = lambda x: f(x)/f_deriv(x)
    f_divf_deriv = lambda x: ((4374*(x**10)) + ...
        (729*(math.e**x)*(x**10)) - (2916*(math.e**x)*(x**9)) - ...
        (4374*(math.e**x)*(x**8)) - (972*(math.e**(2*x))*(x**8)) + ...
        (3888*(math.e**(2*x))*(x**7)) + (972*(math.e**(2*x))*(x**6)) + ...
        (486*(math.e**(3*x))*(x**6)) - (1944*(math.e**(3*x))*(x**5)) - ...
        (108*(math.e**(4*x))*(x**4)) + (324*(math.e**(3*x))*(x**4)) + ...
        (432*(math.e**(4*x))*(x**3)) + (9*(math.e**(5*x))*(x**2)) - ...
        (162**(math.e**(4*x))*(x**2)) - (36*(math.e**(5*x))) + ...
        (18*(math.e**(5*x)))) / ((f_deriv(x))**2)
```

```
    g3 = lambda x: x - (f_divf(x)/f_divf_deriv(x))
    Nmax = 100
    tol = 1e-10

    x0 = 4
    # [xstar,ier,approx,count] = fixedpt(g3,x0,tol,Nmax)
    # print('the approximate fixed point is:',xstar)
    # print('g(xstar):',g3(xstar))
    # print('Error message reads:',ier)
    # print('number of iterations = ', count)



# question4()

def question5():

    f = lambda x: (x**6) - x - 1
    f_deriv = lambda x: 6*(x**5) - 1

    print("Newton Method:")

    g = lambda x: x - (f(x)/f_deriv(x))

    Nmax = 100
    tol = 1e-10

    x0 = 2
    [xstar,ier,approx,count] = fixedpt(g,x0,tol,Nmax)
    print('the approximate fixed point is:',xstar)
    print('g(xstar):',g(xstar))
    print('Error message reads:',ier)
    print('number of iterations = ', count)
    # print(approx)

    newton_error = [abs(xstar - approx[i]) for i in range(0,count+1)]
    iterations = list(range(1,count))

    Newton = pd.DataFrame(list(zip(iterations, approx, newton_error)))
    Newton.columns = ['Iteration','Estimate','Error']
    print("\nNewton Method Error Table:\n")
    print(Newton)
    print("\n")


    print("Secant Method:")

    Nmax = 100
    tol = 1e-10

    x0 = 2
    x1 = 1
    [xstar,ier,approx,count] = secant(x0,x1,f,Nmax,tol)
    print('the approximate fixed point is:',xstar)
    print('f(xstar):',f(xstar))
    print('Error message reads:',ier)
    print('number of iterations = ', count)
    # print(approx)
```

```

    secant_error = [abs(xstar - approx[i]) for i in range(0,count+1)]
    iterations = list(range(1,count))

    Secant = pd.DataFrame(list(zip(iterations, approx, newton_error)))
    Secant.columns = ['Iteration','Estimate','Error']
    print("\nSecant Method Error Table:\n")
    print(Secant)
    print("\n")

    plt.clf()
    plt.figure(figsize=(10, 6))
    plt.loglog(newton_error[:-1], newton_error[1:], 'bo-', ...
        label="Newton Method") # need to take off first b/c undefined, ...
        and last to make same length
    plt.loglog(secant_error[:-1], secant_error[1:], 'ro-', ...
        label="Secant Method")
    plt.xlabel(r'$|x_k - \alpha|$', fontsize=14)
    plt.ylabel(r'$|x_{k+1} - \alpha|$', fontsize=14)
    plt.legend()
    plt.grid(True, which="both", ls="--")
    plt.savefig("HW4.5.b.png")


# question5()
```

NOTE: ALL CODE FOR THIS ASSIGNMENT IS ORIGINAL, WRITTEN BY JESSE HETTLEMAN