Lex Luther Project: Summary of Results

Andrew Rozniakowski

Math 448

2/22/16

<u>Problem Statement:</u>

For this project we had to compute the roots of the function (cos(x) – e^(-x^2) +1)(e^(2*x-1) - (2*x^2) – 1/2) (x^5 - 9*x^4 - x^3 + 17*x^2 - 8*x – 8), in order to stop Lex Luther's devious plan. We were given a Euclidean graph where point (0,0) was the Eiffel Tower. We had to find the (x,y) coordinates of each of Lex's lock boxes, where the x position was the roots of the function accurate to 6 decimal places. For each specific root we had to write a computer program for one bracketing method (Bisection, False Position Method, ect.) and one fixed point method (Newton's Method, Secant Method, ect.) to assure accuracy to 14 decimal points. Once we had the root accurately to 14 decimal points we had to do calculations using the Secant and Bisection methods to determine the y coordinate. Once we had both the x and y coordinate we had to determine the lock boxes code which was the last 8 decimal places of our 14 point roots.

<u>Description of Solution Procedure:</u>

For this project I wrote my methods using Matlab. I wrote programs for Newton's Method, Bisection method, False Position method and the Secant method.

- <u>Outline of Newton's method:</u> With Newton's method you begin with an initial guess we will call x for the root of your given function f. You then find the derivative of your function f. Next you evaluate the function and its derivative using your initial guess, we will call fx and fpx respectively. Your new guess becomes x = x – (fx/fpx). You continually do this until a max number of iterations is reached or your new x and old are a specified small distance apart.
- <u>Outline of Secant Method:</u> For the Secant method you begin with a chosen interval (a,b) and a function f. You then evaluate your function f at these 2 points, we will call them fa

and fb respectively. We then start with 2 guesses which are used to create a secant line to approximate our root. The first guess is x = a and the second is x = a – fa*(a-b) /(fa-fb), where the second guess creates a secant line to our function. We then set fb = fa and reevaluate our function at our new x and this becomes our new fa. Once again, we continually do this until a max number of iterations is reached or our new x and old x are a specified small distance apart. We continue to make secant lines of the function f to approximate its root.

- Outline of Bisection Method: We also start this method with an interval (a,b). Once again we evaluate our given function at a, which we will call fa. Then we create a new endpoint which is exactly half of our original interval, which we will call p. We then evaluate our function at p and call it fp. If fa*fp > 0 then a = p and fa = fp. If fa*fp < 0 then we set b equal to p. We then create a new p using our new a and b values. We continually do this until a max number of iterations is reached or our new x and old x are a specified small distance apart. In this method we are cutting our interval in half over and over again until we approach our root.
- Outline of False Position Method: This method is similar to the Bisection method but instead of just cutting our interval in half every time we try and cut it even smaller with each iteration. Once again we start with an interval (a,b) and we evaluate f at these points to get fa and fb. Instead of cutting it in half like the bisection method we create a p = b – a*(b-a)/(fb-fa). We then evaluate f at p to get fp. If fp*fb < 0 then we set a = b and fa = fb. We also set b = p and fb = fp. Again, we continually do this until a max number of iterations is reached or our new x and old x are a specified small distance apart. This

method makes a more educated guess to where the root is than just cutting the interval in half like the bisection.

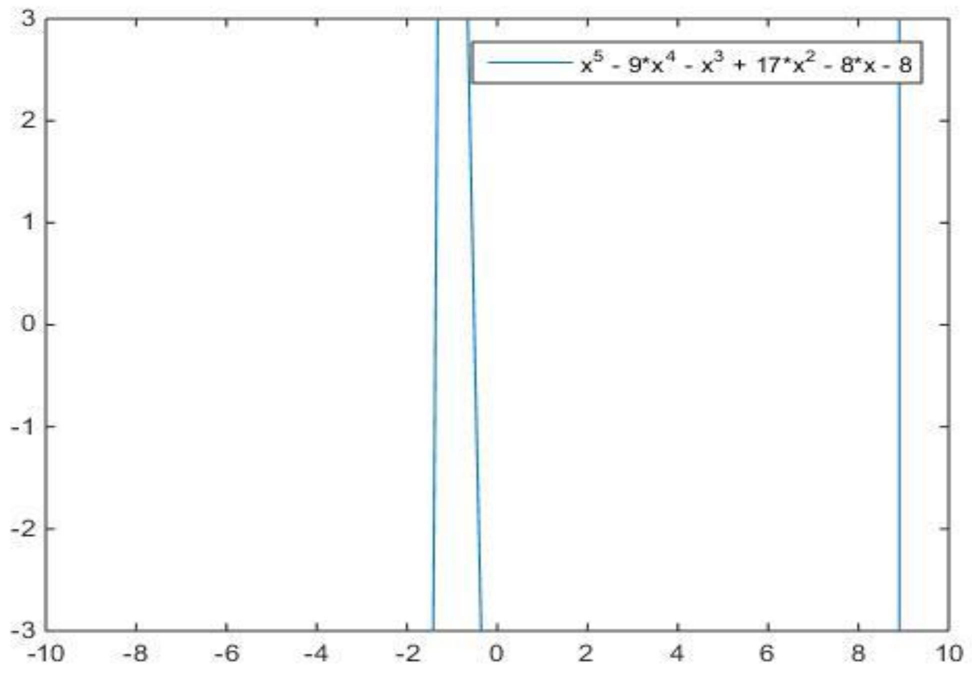
<u>Conclusion Section:</u>

I started this project by splitting up Lex's function into each of its three parts. I started by finding the roots for x^5 - 9*x^4 - x^3 + 17*x^2 - 8*x – 8.

1. I found the first root for this by using the False Position Method as well as the Secant Method. My results were 8.91069640296030 and 8.91069640296030 respectively. For the False Position Method I used the interval [8,9]. I determined this by looking at the graph below. The graph shows it crosses the x axis almost exactly between 8 and 10. Using this interval it took 11 iterations. For the secant method I used x_1 = 8 and x_2 = 9 for the same reason as the false position method. In this case it only took 6 iterations. For each method I used a stopping criteria of when an iteration minus its previous iteration is less than or equal to 1*10^-14 or when 100 iterations were reached. I chose these values as my stopping point because I wanted to assure an answer accurate to 14 decimal places, so I had to have my max iterations at a number I wouldn't expect to reach. As we expected the Secant Method has a faster convergence than the False Position Method.
2. I found the second root for this by using the False Position Method as well as the Secant Method. My results were -1.38750710558262 and -1.38750710558262 respectively. For the False Position Method I used the interval [-1.5,1.5]. I determined this by looking at the graph below. I tried to approximate where the graph crosses the x axis which seemed to be -1.5. Using this interval it took 10 iterations. For the secant method I used x_1 = -1.5 and x_2 = 1.5 for the same reason. In this case it took 7 iterations. As we expect the Secant Method converges faster. For each method I used a stopping criteria of when an

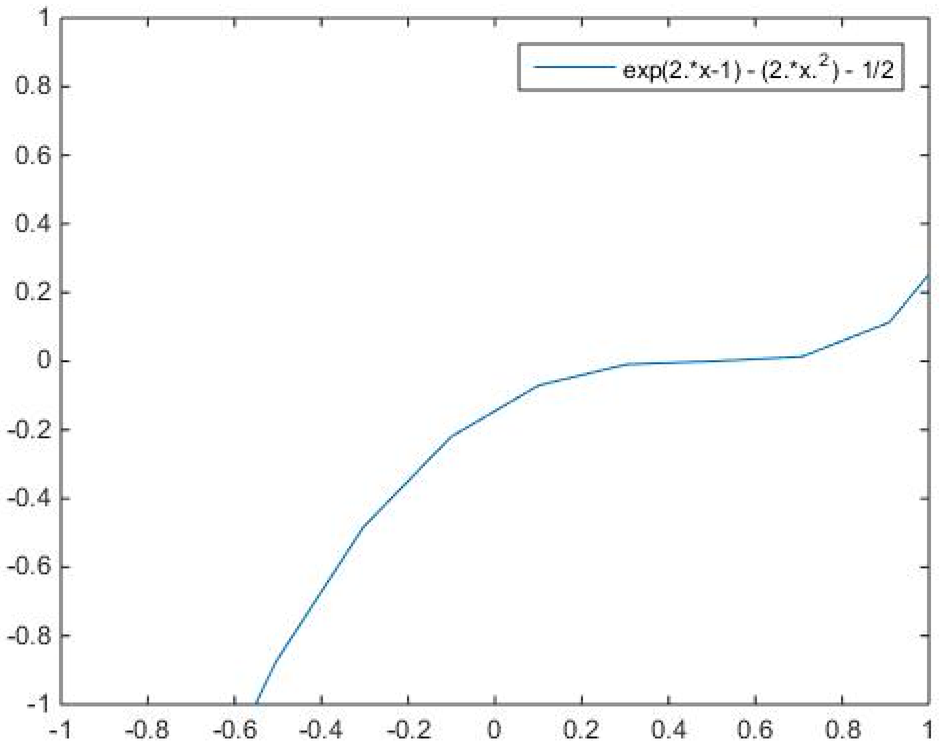
iteration minus its previous iteration is less than or equal to 1*10^-14 or 100 iterations were reached. I chose these for the same reasons as the first root.

3. I found the final root for this part of the equation by using the False Position Method and the Secant Method. My results were -0.51042934281782 and -0.51042934281782 respectively. For the False Position Method I used the interval [-1.3,1.3]. I determined this by looking at the graph below and from the root found in number 2. I wanted to use an interval that captured the last root but I also didn't want to include the root from number 2 because then it might converge to the same root as number 2. Using this interval it took 7 iterations. For the secant method I used x_1 = -1.3 and x_2 = 1.3 for the same reason. In this case it took 9 iterations. In this case we see the False Position Method converge faster, so we lost some convergence for the Secant method. For each method I used a stopping criteria of when an iteration minus its previous iteration is less than or equal to 1*10^-14 14 or 100 iterations were reached. I chose these values for the same reasons as the other roots.



Next I found the roots for e^(2*x-1) - (2*x^2) – 1/2;

1. To find this root I used the Bisection and Secant Method. My results were 0.49999694824219 and 0.50000486762652 respectively. For both methods I chose the interval [-0.6, 0.6]. I determined this interval by looking at the graph below and choosing an interval that captures the root. For the Secant method I used 1*10^-14 or 200 iterations as my stopping criteria. I did this to assure accuracy to 14 decimal places and because I noticed it was taking more than 100 iterations to assure this accuracy. Using this stopping criteria I had 104 iterations. I used the same criteria for the Bisection Method, however it only took 16 iterations, which means it hit my max iteration limit before converging to 14 decimal places. I chose the bisection method because it worked better than the false position method in this case. Surprisingly, the bisection method converges must faster.



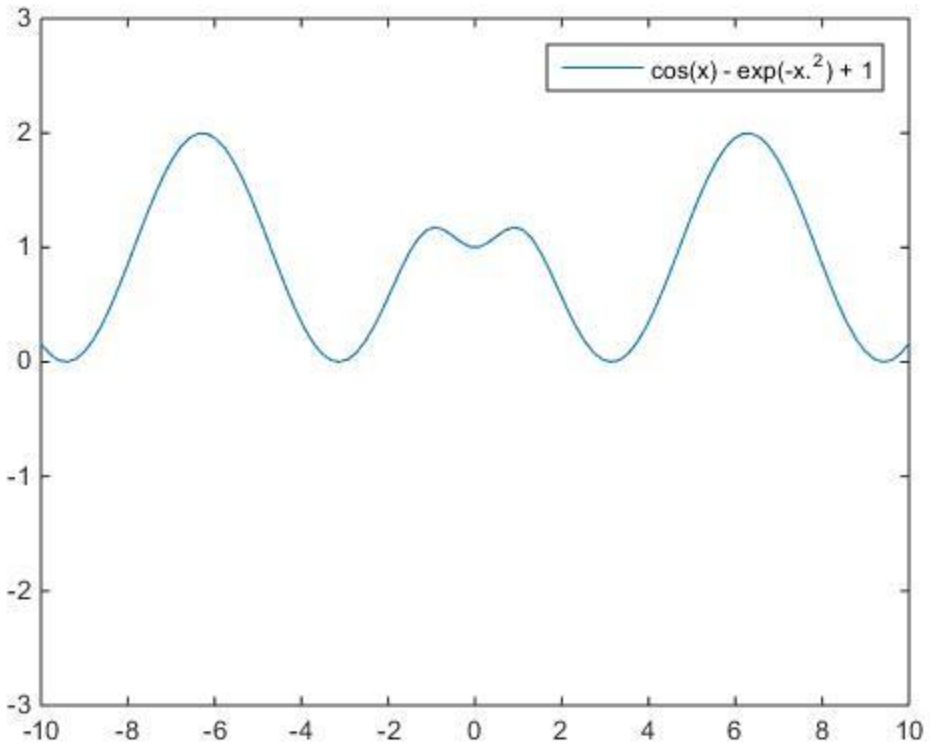
Finally I found the roots for: cos(x) – e^(-x^2) + 1

1. For this part of the function I used Newton's method and the False Position Method. I obtained the results 3.15145285136407 and 3.15145304883031 through my methods respectively. For Newton's method I used an initial guess of 3.5 because I tried to approximate the root looking at the graph below. I also used a max iteration of 100 and a stopping criteria of 1*10^-4. I used a smaller stopping criteria for newton's method because increasing it made the program run exponentially slower so I tried to optimize the speed of the program with its accuracy to 14 decimal places. It took 8 iterations using this initial guess. Similarly I used the interval [3.5, 4] for my false position method to try and capture the root in my interval. This took 201 iterations

meaning it hit its max iterations of 200. As expected, Newton's method seems to be converging quadraticly and much faster than the False position method.

2. For the next root I also used newton's method and the false position method. I obtained the results 3.13108083839013 and 3.13108083857001 respectively. For newton's method I used the same stopping criteria but this time I used an initial guess of 3. I determined this by looking at the graph below and approximating the root as well as trying to pick a guess that's closer to a new root than the root found in 1. Using this initial guess it took 6 iterations. For the false position method I used the same stopping criteria but changed the interval to [3,4] for the same reason I chose newton's initial guess to be 3. The false position method took 184 iterations this time.
3. Once again I used Newton's method and false position method and obtained the results -3.13108083839013 and -3.13108083857001. For newton's method I used the same guess as part 2 but I changed it to -3 to get the negative result. It took 7 iterations. For the false position method I used the interval [-3,-1]. I determined this by looking at the graph below and assuming the root would be approximately -3.13 based on the positive root, so I chose an interval that would be closer to -3.13 than -3.15, the other root I'm expecting. This method took 184 iterations. So once again Newton's is much faster.
4. For the last root I again used Newton's method and false position method and obtained the results -3.15145285136407 and -3.15145363886255. Just like the third root I found, I used my initial guess from part 1 but flipped it to the negative value to find this root using Newton's method. It took 8 iterations. For the false position method I used the interval [-4, -3.5] once again so I would be closer to the root -3.15

than -3.13. I also used a max iteration of 400 in this case for the false position method to assure as much accuracy as I could. We see in this situation how much faster Newton's is compared to the false position method.



Anytime the 2 methods did not match I took average of the 2 before completing the rest of the project. I felt the most difficult part of this project was trying to get the accuracy of the roots to 14 decimal places, especially using matlab. I felt matlab was not well suited for this and the roots would often change drastically in the 14 digits after the decimal even when only minimal changes were made.

<u>Final Results:</u>

Results of bisection iterations:

Results of Bisection Iterations	Results of Secant Iterations
8.8272	9.6383
3.2745	3.1515
5.9435	3.1311
0.6553	0.4704
4.1716	-0.5104
0.4272	-1.3875
-0.9450	-3.1311
-1.7931	-3.1732

Final (x,y) coordinates:

(8.910696, -0.209464) Go 891.069640 miles east and 20.946366 miles south

With the Lock Box Code: (4,0)(2,9)(6,0)(3,0)

(3.151453, -2.094838) Go 315.145295 miles east and 209.483769 miles south

With the Lock Box Code: (9,5)(0,0)(9,7)(1,9)

(3.131081, 1.034028) Go 313.108084 miles east and 103.402850 miles north

With the Lock Box Code: (8,3)(3,1)(1,5)(0,2)

(0.500001, -1.688308) Go 50.000091 miles east and 168.830824 miles south

With the Lock Box Code: (9,0)(7,9)(3,4)(3,6)

(-0.510429, 1.543730) Go 51.042934 miles west and 154.372962 miles north

With the Lock Box Code: (3,4)(2,8)(1,7)(8,3)

(-1.387507, 0.595921) Go 138.750711 miles west and 59.592127 miles north

With the Lock Box Code: (1,0)(5,5)(8,2)(6,2)

(-3.131081, 0.782119) Go 313.108084 miles west and 78.211886 miles north

With the Lock Box Code: (8,3)(3,3)(4,9)(3,1)

(-3.151453, 0.322148) Go 315.145325 miles west and 32.214754 miles north

With the Lock Box Code: (2,4)(5,1)(1,3)(3,1)