Austin Dollar

CSCI 551

Assignment 6

1) [25 points total] Polynomial function root solving (degree 2 or higher).

a. [5 pts] Using a cloud math tool (e.g., Desmos), Excel, MATLAB, or Gnuplot (or graph paper by hand), plot the following function - x 3 + 9.7x 2 − 1.3x − 105.7 and determine where it has roots (X-axis zero crossing) by observation (you can check answers with MATLAB too). Use the plot to get your starting guesses for the Newton Raphson root solver. Note that you will need to compute the derivative as $d\ dx$ (−x 3 + 9.7x 2 − 1.3x − 105.7) and create an updated version of Newton Raphson starter code (here or other starter code you find or your own implementation). Provide your plot, derivate and updated code.

Graph Via Desmos:

When attempting to solve roots via observation, confirmed via desmos' zero calculator, we can confirm that the roots are: **-2.8, 4.7, 7.7**

The calculated derivative of the given equation, is as follows : $-\ 3x^2\ +\ 19.4x\ -\ 1.3$

The code with both f(x) and d(fx) is titled newton.c, included in the Q1 folder of the attached code.

b. [5 pts] Find all of the roots of $-x$ 3 + 9.7x 2 − 1.3x − 105.7 using your Newton-Raphson code, your guess(es) for roots, your derivative function, and your iteration parameters, and then compare your results to solutions determined using MATALB and/or by testing (plugging results back into the original equation) and estimate the error in your solution. Work to get less than 0.1% error. Please check your work with online math tools and plug your root back into the equations to see how close you are to having a true zero for the function!

The solution for my guesses for roots, as well as the derivative function, and my iteration parameters are included below:

My root guesses: -2.8, 4.7, 7.7

The derivative equation: $-\ 3x^2\ +\ 19.4x\ -\ 1.3$

My iteration parameters: -2.8, 1, 100

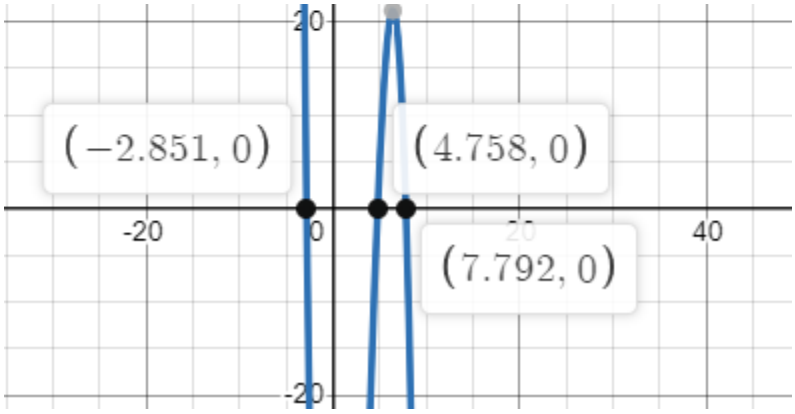
4.7, 1, 100

7.7, 1, 100

Output of Newton-Raphson Code:

```
addollar@o244-11: ~/551/HW6/Q1
addollar@o244-11:~/551/HW6/Q1$ ./a.out

Enter x0 guess to find first root, allowed error and maximum iterations
-2.8 1 1
Iteration:   1, x = -2.851301
After   1 iterations, root = -2.851301
addollar@o244-11:~/551/HW6/Q1$ ./a.out

Enter x0 guess to find first root, allowed error and maximum iterations
4.7 1 1
Iteration:   1, x =  4.757603
After   1 iterations, root = 4.757603
addollar@o244-11:~/551/HW6/Q1$
addollar@o244-11:~/551/HW6/Q1$ ./a.out

Enter x0 guess to find first root, allowed error and maximum iterations
7.7 1 1
Iteration:   1, x =  7.796341
After   1 iterations, root = 7.796341
addollar@o244-11:~/551/HW6/Q1$
```

Solutions according to Desmos:



Estimated Error(using output from above code output and desmos):

Error for 2.8 root: 0.01%

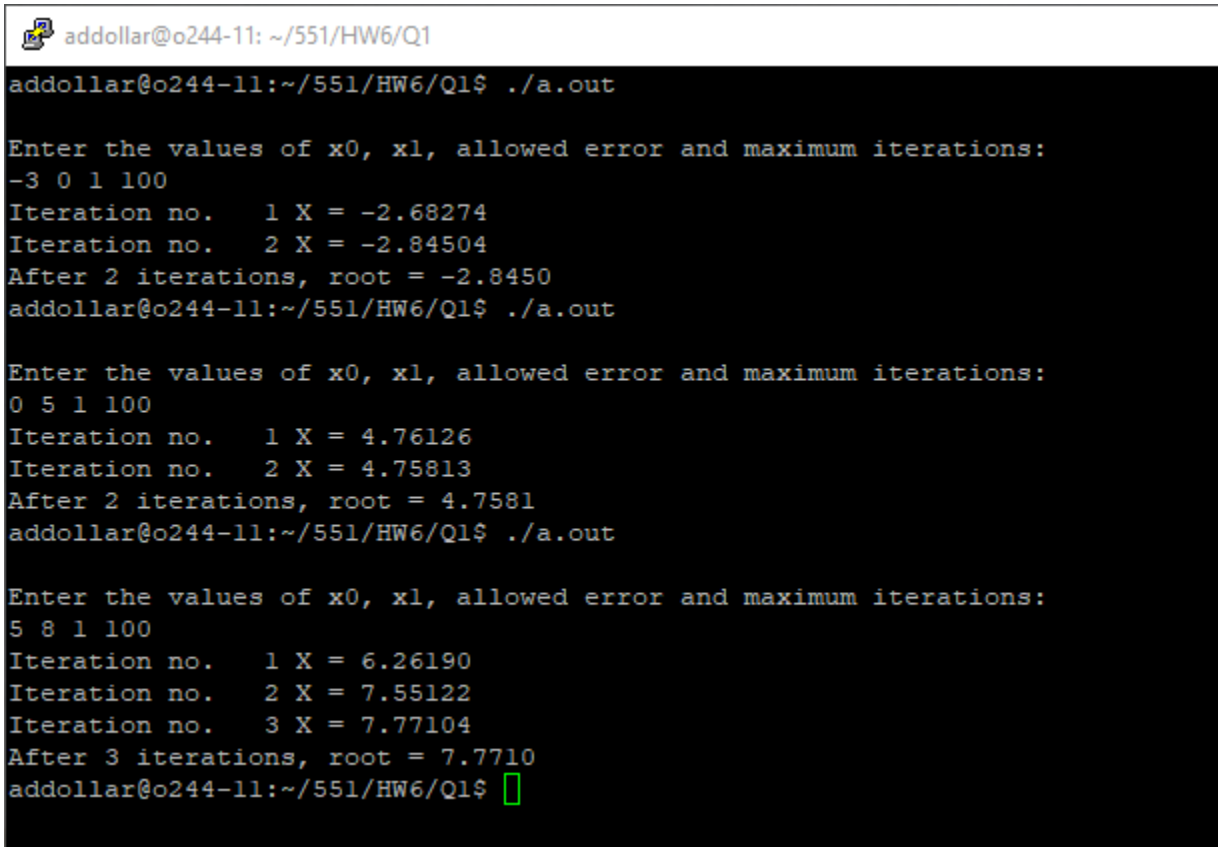
Error for 4.7 root: 0.008%

Error for 7.7 root: 0.05%

Average percent error: 0.02%

c. [5 pts] Now find all of the roots of $-x$ 3 + 9.7x 2 − 1.3x − 105.7 using Regula Falsi (e.g. adapt this starter code, or code yourself). Again, attempt to find roots with error less than 0.1%.

Output of Regula Falsi (included in Q1 folder as regulafalsi.c):



Estimated Error(using above desmos outputs with Regula Falsi outputs):

Error for -2.8 root: 0.21%

Error for 4.7 root: 0.002%

Error for 7.7 root: 0.26%

Average Percent Error: 0.15%

d. [5 pts] Compare your Newton Raphson to Regula Falsi for real valued roots of polynomials (note that if you have a negative discriminant, you will have complex roots) in general, and report on which algorithm converges to the same error target faster (fewest iterations). You can do this for one root or all roots on an interval, but you must be sure to use similarly informed guesses and error tolerances for your comparison.

When observing the two types of root solving, the Newton Raphson converged much faster. It took very few iterations to produce very little error in the produced roots (based on the desmos values), with a percent error averaging at 0.02%. Meanwhile, the Regula Falsi method, while it is relatively accurate, took many more iterations to produce a not as accurate solution, with average error pushing 0.15%.

e. [5 pts] Time your Newton Raphson and Regula Falsi sequential programs using POSIX clock_gettime for the same polynomial, with the same guesses for roots and error tolerances. Compare the computation times.

Output of Newton Raphson:

```
Enter x0 guess to find first root, allowed error and maximum iterations
-2.8 1 100
Iteration:   1, x = -2.851301
After   1 iterations, root = -2.851301
Total time = 6.640972 seconds
addollar@o244-11:~/551/HW6/Q1$
```

Output of Regula Falsi:

```
Enter the values of x0, x1, allowed error and maximum iterations:
0 3 1 100
Iteration no.   1 X = 5.62234
Iteration no.   2 X = 4.88764
After 2 iterations, root = 4.8876
Total time = 7.648212 seconds
addollar@o244-11:~/551/HW6/Q1$
```

Using POSIx gettime, it seems that Newton Raphson is faster, as well as more accurate, as previously stated, being almost a full second faster than the Regula Falsi method.

2) [25 points total] Transcendental function root solving where derivative is unknown and/or hard to estimate.

a. [5 pts] Consider the functions $-1.5x + 2.5 + \sin(x\ 2)$ and $\cos(x\ 2)$ as shown below on the interval [0.0,10.0].

{image on pdf}

Find the first real root for the equations above on the interval [0.0, 2.0].

The first real root for these equations is: 1.25 (found via observation of the given graph)

b. [5 pts] Find roots for the intersection of the 2 functions above on the interval [0.0, 2.0]. Note that you should use F(x)=f(x)-g(x) or $F(x) = \cos(x\ 2) - (-1.5x + 2.5 + \sin(x\ 2))$ and find where this difference function is zero to find the crossing on the interval [0.0, 2.0].

The root of the difference function on [0,2] is as follows: 1.75

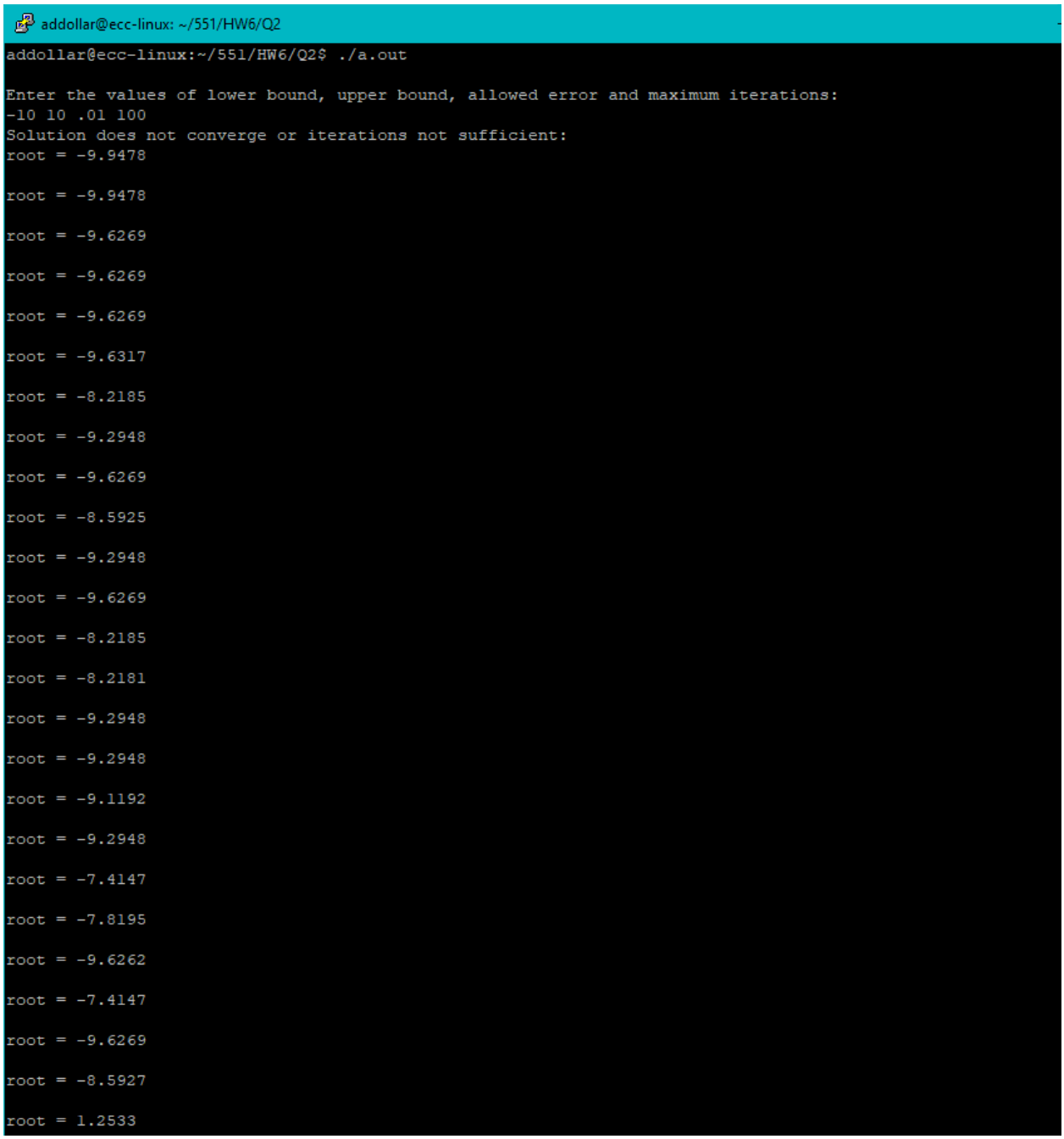
c. How many times does $\cos(x\ 2)$ cross the X-axis (how many zeros does it have) on the interval [0.0, 10.0]? Please answer based upon enhancing your program so it has the ability to find roots more than once, looping through your root solver multiple times, but also compare to your expectations as well (the Excel graph is here). Note that a system for guessing where you use the last root found plus a small offset based on sign change will help give the periodic nature.

I modified the regula falsi code in order to accommodate solving for multiple roots. I did so by turning the old main into just a normal function, with it being called in a loop. The loop will slowly change the x0 and x1 parameters as we move through the loop. Doing so, however, resulted in a lot of NA values for some reason and some returned values that are out of scope. This was fixed with an if statement that only prints roots if they are within our initial scope. While not perfect, as it did not get all of the roots, it still got some of them:

```
addollar@ecc-linux: ~/551/HW6/Q2
Enter the values of lower bound, upper bound, allowed error and maximum iterations:
0.0 10.0 0.001 1000
Solution does not converge or iterations not sufficient:
root = 5.7434

root = 2.8025

root = 1.2533

root = 2.8025

root = 1.2533

root = 1.2533

root = 1.2533

root = 1.2533

root = 1.2533

root = 1.2533

root = 1.2533

root = 1.2533

root = 1.2533

root = 1.2533

root = 1.2533

root = 2.1708

root = 3.7599

root = 5.1675

root = 2.8025

root = 1.2533

root = 1.2533

root = 1.2533

root = 1.2533

root = 1.2533

root = 1.2533

root = 5.1675

root = 8.7732

root = 1.2533

root = 1.2533

root = 1.2533

root = 2.8025
```

d. Adapt your program for general root finding, without a derivative function, so that it can find any number of roots on any interval in theory, and test it specifically for the interval [-10.0, 10.0] breaking the problem down into smaller intervals if needed, and show that it works for cos(x 2). For example, see the plot of the two functions we are studying here.

The same file, regulafalsiseq.c works for general root finding as well, as shown below on the interval [-10, 10]:



e. [5 pts] Take your program above and add OpenMP directives to make parallel and compare the speed-up with purely sequential for cos(x 2) and compare on a larger interval such as [-100.0, 100.0].

The new, parallel version of the code is included in the Q2 folder entitled rfpar.c. Below is the posix timed output of the sequential program, followed by the parallel program posix time:

Sequential:

```
root = -31.9288

Total time = 0.057159 seconds
addollar@ecc-linux:~/551/HW6/Q2$
```

Parallel:

```
root = -31.9288

Total time = 0.017760 seconds
addollar@ecc-linux:~/551/HW6/Q2$
```

Overall, the calculated speedup is a factor 0f 3.2184

3) **[50 points total] The following problem involves two trains, one that uses a linear acceleration controller and one that uses non-linear (found here). The trains leave the same station, are the same length, and run on parallel tracks. Plotting the train positions as a function of time, it appears the 2 trains will be lined up at the starting station, once during the trip, and again at the destination station.**

a. [10 pts] Write a sequential C/C++ program to integrate using your preferred method (Riemann, Trapezoidal, other) using double precision and with accuracy that is ideally off by no more than 1 meter (1 meter / 122,000 meter suggests no more error than 1 ppm, and at least 15 digits of precision). Because the acceleration functions are both designed to have zero velocity at the end of the trip, this is a good check for accuracy. You will have to create a look-up-table with interpolation or a piecewise linear function and transcendental function for acceleration for each. Provide error analysis by comparing the velocity of each at the end of 1,800 seconds.

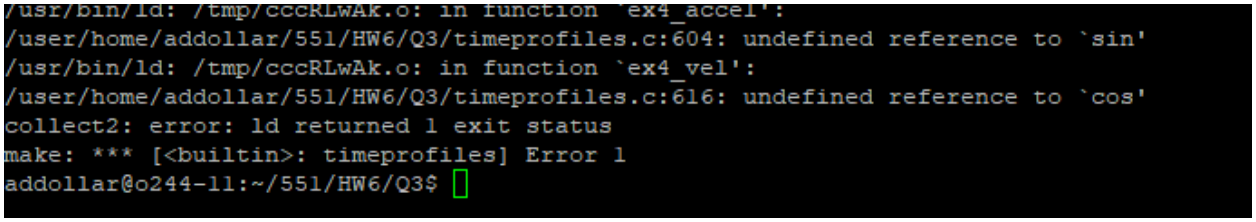
b. [10 pts] Now speed-up your program above as best you can using OpenMP block pragmas in your code, and time your simulation for the interval [0, 1800] seconds using POSX clock_gettime and compare the sequential to the OpenMP parallel.

Combining parts a and b, we have the single threaded and multithreaded version with their given timed outputs below:

```
SINGLE THREAD: Left Riemann sum test for table with 1801 elements
Train from table in 0.000261 seconds with 1801 samples: final velocity = 0.000000, final position = 122000.000000


THREADED FOR LOOP: Left Riemann sum test for table with 1801 elements
Train from table in 0.008225 seconds with 1801 samples: final velocity = 0.000000, final position = 122000.000000
```

c. [15 pts] Use either of your two programs above to determine the time after the train starts when the trains are lined up again before arriving at their destination (when the trains are lined up during the trip).
d. [15 pts] Use either of your two programs above to determine the position in meters during the trip where the two trains are lined up (at the same position).

For parts c and d, I ran into some weird issues. When compiling, even though I have the math library included, it still gave me the error that sin and cos are undefined. I went and tried many different debugging strategies, but nothing seemed to work. Below is a screenshot of the error I kept getting:


/usr/bin/ld: /tmp/cccRLwAk.o: in function `ex4_accel':
/user/home/addollar/551/HW6/Q3/timeprofiles.c:604: undefined reference to `sin'
/usr/bin/ld: /tmp/cccRLwAk.o: in function `ex4_vel':
/user/home/addollar/551/HW6/Q3/timeprofiles.c:616: undefined reference to `cos'
collect2: error: ld returned 1 exit status
make: *** [<builtin>: timeprofiles] Error 1
addollar@o244-11:~/551/HW6/Q3$