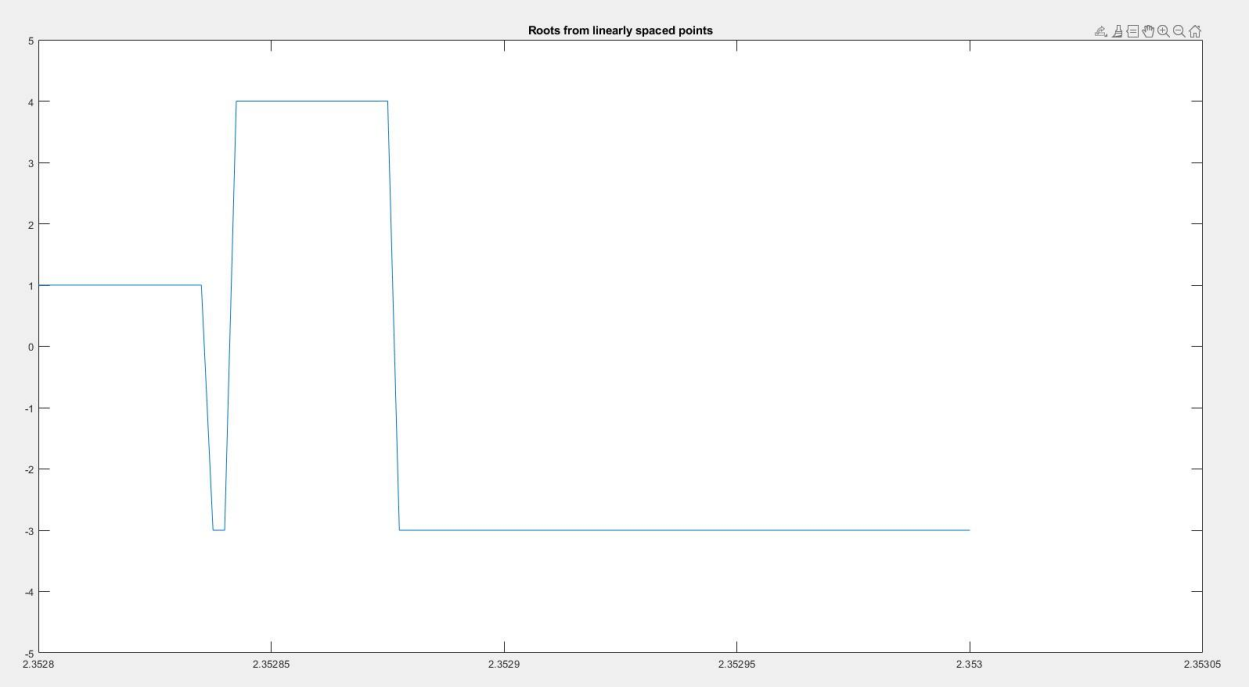
Brandon Walters

CS 3200

Prof. Berzins

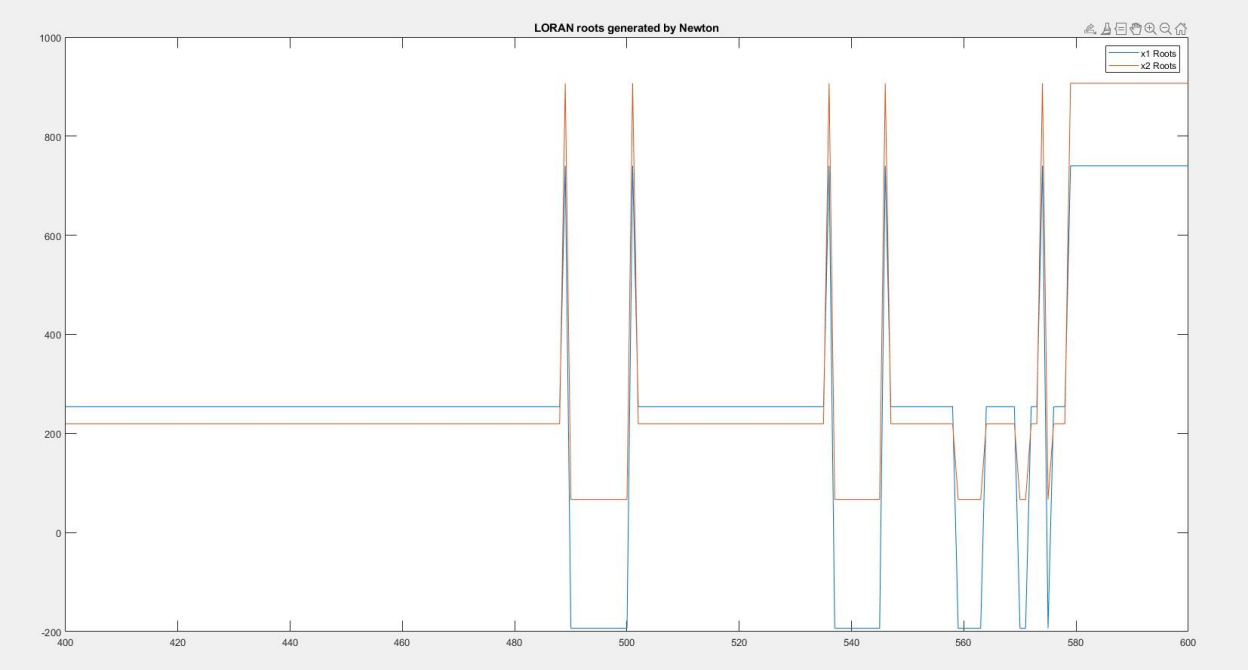
Homework 7 Report

1. As we can see from the rootCheck calculation that utilizes matlab's roots() method, the roots of our function are indeed 4, -3, and 1. The claim by University X is correct, as we can see from our function's Newton method work when given the university's points. I found that although 20 iterations of Newton's method did produce strong leanings towards a convergence and full convergence for some of the starting points, 30 iterations produced full convergence for all given starting points, which is why I specifically used 30 iterations of Newton's method for my function. As we can see from the printout, the first and last points give a root of 1, the middle point gives a root of 4, and the other two starting points converge to -3. Using 81 linearly spaced points for another iteration of Newton's method, we can see on our graph that the convergence differs quite a bit depending on the starting point. Typically the root 1 is only converged with lower starting points, 4 only for a certain section around 2.35285, and -3 for larger starting values. A cleaner set of points that all converge to each of the different roots would be x = [2.3528, 2.35285, 2.353]. This pointset accomplishes the same task of the University X points of converging to the different roots, while being both shorter and much less precise, making them easier to read, input, and interpret.

2. 

3. For the LORAN equations, we can define the Jacobian matrix as [x(1)/17298 -x(2)/27702; -2*(x(1)-300)/172159 2*(x(2)-500)/77841], which allows us to properly implement Newton's method for these equations. If the solution reaches (0,0) or (300,500), portions of the Jacobian matrix will be zeroed out, as these values being inputted into the matrix will create 0 values in our Jacobian matrix. Running our Newton's method for 25 iterations on points from (400,400) to (600,600), we can see from the graph made from the answers that there are 3 roots for both x1 and x2, for a total of 6 different root solutions. For pairs from (400,400) to around (475,475), the roots converged to for x1 and x2 are 254.221 and 219.307, respectively. For small sections, such as around (495,495), the roots are -193.295 for x1 and 66.565 for x2. Finally, the most consistent section for the third and final pair of roots can be found at the high end of the spectrum, around (590,590). The roots converged to here are 740.329 for x1 and 906.826

for x2. Like Question 1, we can observe that the choice of starting point does drastically affect the converged values, with various tiny spikes and changes for certain point pairs, such as (489,489) and (501,501).

4. 

5. Using Simpson's rule on the function x^p across the interval [0,1] for different exponential values shows us an interesting development. For each set of points, spanning from 17 all the way up to 513, we can see from the first table that Simpson's actually converges to the same values for every point set. This shows that even at a low amount of points, Simpson's Rule provides us with a consistent and useful tool to be able to evaluate integrals. Looking at the second integral and its associated convergence table, we can see that once again our Simpson implementation is able to converge to a common value on all point sets, that common value being 7.0523. As we can see from the timing column on the table, we observe that as the point count increases, the time needed to converge using our Simpson implementation also increases. This makes sense, as the

added points mean that more calculations must be performed by the Simpson setup, meaning that the calculations will take longer and therefore it will take longer for us to converge.

6.

	Power	17 Points	33 Points	65 Points	129 Points	257 Points	513 Points
1	2	0.3333	0.3333	0.3333	0.3333	0.3333	0.3333
2	3	0.5833	0.5833	0.5833	0.5833	0.5833	0.5833
3	4	0.7833	0.7833	0.7833	0.7833	0.7833	0.7833
4	5	0.9500	0.9500	0.9500	0.9500	0.9500	0.9500
5	6	1.0929	1.0929	1.0929	1.0929	1.0929	1.0929
6	8	1.2040	1.2040	1.2040	1.2040	1.2040	1.2040

7.

	Number of Points/Iterations	Common Value of Convergence	Time to Converge
1	17	7.0523	2.9200e-05
2	33	7.0523	1.1700e-05
3	65	7.0523	2.1100e-05
4	129	7.0523	4.0800e-05
5	257	7.0523	8.0400e-05
6	513	7.0523	1.6000e-04

8. We can see that when using QuadTX() with different levels of tolerance, our number of iterations increases in a roughly linear fashion in response to the stricter levels of tolerance, starting at 813 iterations and climbing up to 19,545 iterations in the 1e-14 tolerance case. In coordination, we can also observe that the time taken by the entire QuadTX function to converge also increases as the tolerance climbs, going from 0.012 seconds to a maximum of 0.137 seconds for the 1e-14 case. Overall, we can see that increasing the tolerance leads to a roughly linear corresponding increase in both the number of iterations needed to converge and the total time needed to run the QuadTX() function.
9. Comparing this integral when evaluated by the normal Simpson code and the QuadTX() function yields some interesting results. Obviously because the normal Simpson's code only runs based on the certain number of points provided (here

using the array for earlier questions, from 17 all the way up to 513), even the smaller tolerances for the QuadTX will use many more iterations to converge, a fact supported by the respective timing arrays, where even the fastest QuadTX time of 0.016 seconds is slower than 0.0015, which is the normal Simpson's slowest time. Overall, we can see that the normal Simpson's code runs much faster overall than the QuadTX function, although the QuadTX recursive version of Simpson's is much more accurate and can operate at generally higher levels of tolerance more acceptably.