Jared Rivera

804603106

CEE/MAE M20

March 7, 2017

Homework 8

1 The Ranked-Choice Vote

1.1 Introduction

The goal of this program is to simulate the results of an election in a system that used ranked-choice voting. Ranked-choice voting allows voters to rank each candidate from most to least preferable. The program checks to see if any candidate has a clear majority (meaning more than 50% of the total vote) and if there isn't one it removes the least popular candidate from the ballot. It then goes back through and checks for a majority, continuing this iteration until there is a clear victory. The program then declares the winner, as well as the ballot history throughout its iterations and how long it took to find a winner.

1.2 Model and Methods

The first step in this program is to call load the voting data provided in the problem statement and initialize the number of voters, candidates, and initialize the matrices that will hold the number of votes each candidate receives and the vote history to be printed at the end. The final variable to be initialized is the count variable to be used to determine the number of iterations needed to find a winner. A while loop is then entered that will run as long as there is no clear winner, the logic of it being shown below.

```
max(VoteCount)<=(sum(VoteCount)/2)
```

Inside the loop, the count for number of iterations is updated by 1 and the present state of the election is assessed. The number of total votes is calculated, as well as the number of candidates using the size function. An array that will total the votes for each candidate is then initialized and populated using the method shown below.

```
    for k=1:voters
  for j=1:NumCandidates
    if votes(k,1)==j
VoteCount(j)=VoteCount(j)+1;
            end
            end
            end
```

The array is then sorted to determine the current winning and losing candidates, and the matrix holding voting history is updated. Now the removeCandidate function is called.

The removeCandidates function initializes a new matrix to hold voting data and populates it using three for loops. The first two iterate through the matrix in the standard procedure, and the last iterates in the case that the losing candidate is encountered. In that situation the entry is removed and the row is shifted to the left. Otherwise, the entries form the original matrix are copied, as shown on the following page.

```
if votes(j,k)==losingCandidate
    for l=k:NumCandidates-1
  votesnew(j,l)=votes(j,l+1);
                end
    elseif votesnew(j,k)==0
   votesnew(j,k)=votes(j,k);
                end
```

Finally, the old voting matrix is reassigned as the new one and passed back through the function until the while loop ends, and the results of the election are printed to the screen.

1.3 Calculations and Results

There are two possible cases for this program, one for each sets of votes. When the program is run using votes1, the following is printed to the command window

```
                  1     2   3   4   5   6   7 8
Round 1 Totals: 2012 253 240 1998 242 103 92 60
Round 2 Totals: 2022 264 246 2010 252 108 98 0
Round 3 Totals: 2038 281 265 2029 269 118 0 0
Round 4 Totals: 2060 305 285 2055 295 0 0 0
Round 5 Totals: 2128 369 0 2135 0 0 0 0
Round 6 Totals: 2128 369 0 0 0 0 0 0

Winning Candidate: 1
```

When the program is run using votes2, the following is printed to the command window

```
1     2   3    4   5    6   7  8
Round 1 Totals: 2272 368 365 3731 395 141 156 72
Round 2 Totals: 2280 371 375 3746 408 151 169 0
Round 3 Totals: 2304 393 401 3773 435 0 0 0
Round 4 Totals: 0 0 0 0 0 0 0 0
Round 5 Totals: 0 0 0 0 0 0 0 0
Round 6 Totals: 0 0 0 0 0 0 0 0

Winning Candidate: 4
```

1.4 Discussion

As can be seen from looking at the results section, I hard coded 6 rounds into the display because I had some problems with variable iteration in a loop that will print out the least amount of rounds necessary. Regardless, the display is fairly clean. In the two cases presented above, there is a clear candidate to eliminate each round. This is the case because no one tied for the least amount of votes. In the case that there's a tie for elimination, this code won't work. To account for that, an edit can be made in the form of analyzing the further column of the ballot data. For example, if candidates 1 and 2 tie for the least amount of primary votes, the program can then look through the secondary and tertiary rankings until it finds one of the two that is the true loser (harsh I know). This can be done using similar framework to the vote count portion of the main script.

Another method for voting can be done in the form of weighted voting, where the first place gets 4 points, second gets 3 points, etc. In this system, the results are much different. For votes1, candidate 8 wins. For votes2, candidate 8 also wins. This may have some implications to the fairness of the ranked-choice voting system, in that it doesn't account for where the candidates are ranked past the primary position. For example, the example shown in lecture showed the candidate that dominated the second place position being eliminated first. This may mean that a mixture of weighted and ranked-choice systems may be more fair in taking into account the voice of the voters as a whole.

2 Newton's Method

2.1 Introduction

The goal of this program is to use perform Newton's method for finding roots of a function. The mathematical model is derived from the derivative and can easily track how many steps are needed to find a zero to a certain precision. This model is passed into the program, which analyzes a specified function over a specified domain and steps through the x-values, determining how many steps it takes to find the root, as well as its location, at each step. The program to the command window the starting x-value at each step, the number of iterations of Newton's method needed to find a zero as well as its location.

2.2 Model and Methods

The first step in this program is to determine the number of steps needed to step through the entire x-range, and to also set an array with the x-range. Next, the parameters of delta and the maximum number of evaluations through Newton's method are set. The loop that goes through each x-value in the range is then entered, where the Newton function is called and its outputs and the x0 value are printed in the manner specified in the problem statement.

In the Newton function, the first step is to set a count variable that will assure the method is only run less times than the maximum number previously specified. Next, the while loop is entered with the logical criteria shown below.

```
(abs(equation(x0))>delta) && (count<fEvalMax)
```

Inside the loop, Newton's method in terms of the equation is applied, calling the specified function it's performed on in the process, as well as the central differentiation function written in week 5, just updated to take in the equation in this script. The Newton's method equation is shown below.

```
xc=x0-(equation(x0)/df_central(@equation,x0));
```

The counter is then updated and the current x position is set to the x0 value for the next iteration. Finally, the number of evaluations is outputted to the main script and the final results of the program are printed, as shown below.

2.3 Calculations and Results

When the program is run, the following is printed to the command window.

```
x0=1.430000 , evals=4, xc=1.470588
x0=1.440370 , evals=4, xc=1.470588
x0=1.450741 , evals=4, xc=1.470588
x0=1.461111 , evals=3, xc=1.470588
x0=1.471481 , evals=2, xc=1.470588
x0=1.481852 , evals=3, xc=1.470588
x0=1.492222 , evals=4, xc=1.470588
x0=1.502593 , evals=6, xc=1.470588
x0=1.512963 , evals=7, xc=1.666667
x0=1.523333 , evals=4, xc=1.562500
x0=1.533704 , evals=3, xc=1.562500
x0=1.544074 , evals=3, xc=1.562500
x0=1.554444 , evals=2, xc=1.562500
x0=1.564815 , evals=2, xc=1.562500
x0=1.575185 , evals=2, xc=1.562500
x0=1.585556 , evals=3, xc=1.562500
x0=1.595926 , evals=3, xc=1.562500
x0=1.606296 , evals=4, xc=1.562500
x0=1.616667 , evals=5, xc=1.470588
x0=1.627037 , evals=8, xc=1.666667
x0=1.637407 , evals=5, xc=1.666667
x0=1.647778 , evals=4, xc=1.666667
x0=1.658148 , evals=3, xc=1.666667
x0=1.668519 , evals=2, xc=1.666667
x0=1.678889 , evals=3, xc=1.666667
x0=1.689259 , evals=4, xc=1.666667
x0=1.699630 , evals=4, xc=1.666667
x0=1.710000 , evals=4, xc=1.666667
```

2.4 Discussion

As can be seen by the data in the results section, x0 plays a big part in the position of the zero found by the function. The function plot is included below, and it is evident that Newton's method tends to target zeros that are near it and in front of it. Some exceptions to this are x0=1.51 and x0=1.61. Both of these values find zeros that are a good distance from the current x. This is due to the projection of the next x-position in the iteration along the slope at the current point, which is close to 0 at each of those points. Specifically, when x is in the range of 1.61 to 1.62, the function has to work much harder to produce a quality answer, iterating over twice as many times as the average over the whole data set.

The specified delta value also has impacts on the output of the program. With a larger delta, the program iterates much less due to less of a need for precision in final calculation. The largest delta value that the program will take is 10^{-2}, which on average only requires a couple of iterations, opposed to a small number like 10^{-11} which takes closer to 7 iterations to provide the needed accuracy.