Juan Villaman

05/02/2024

**Plotter, Salter, and Smoother in MATLAB**

# Learning MATLAB and Making the Plotter

Taking on this project, I have practically never worked with MATLAB, but had seen some it around the internet as the years went by and I was learning how to code. I had a rough understanding that it was an application that was able to easily run complicated and lengthy math operations that may be difficult to do by pen and paper.

Since my initial goal is to make these 3 programs, I began on the Plotter. I wanted user input in MATLAB, and I did not know if it was possible. After a quick google search, I learned two things: that MATLAB syntax reads like English, and how to take user input ([Official MATLAB Documentation](https://www.mathworks.com/help/matlab/ref/input.html)). Taking user input take a simple line of code:



Next was figuring out how to generate the X values. This led to me looking for some sort of for-loop within MATLAB. While there are for loops, I found an operator which works way better for this case, which is the [colon (:)](https://www.mathworks.com/help/matlab/ref/double.colon.html) operator. Below is an example of what it looks like:

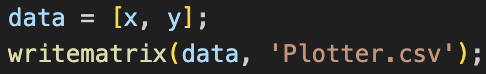


This generates a list of X-values from -20 to 20, incrementing by 1. The number in the middle determines how much x is incremented between -20 and 20. For the purpose of the plotter, we need x to increment by 1. Next, we need to calculate the Y-value for each X. This is pretty simple, as MATLAB writes like English, it is easy to write the quadratic function:

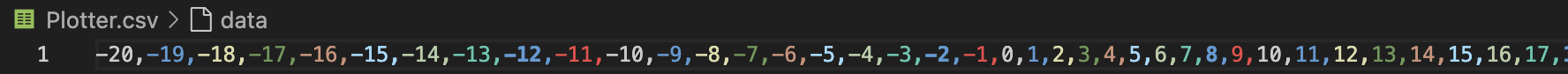
Like this: **y = a \* x \* x + b \* x + c**. However, we need to calculate the Y-value for EACH X-value that is in the range we have designated. There is an operator that allows us to do that for every single X-value. Since x is a vector in this case in MATLAB, we can use element-wise power to calculate each Y-value of the X values ([Source](https://www.mathworks.com/help/matlab/ref/double.power.html)/[Source #2](https://www.mathworks.com/help/matlab/matlab_prog/array-vs-matrix-operations.html)). It looks like this in our program:



The little bit allows us to calculate every Y-value of the X-values for our plotter program. The next step is getting to store those values into a CSV file. Using the writematrix() method, we can write matrices directly into CSV files. At first, I ran into an issue when the first CSV file was created. I created a matrix called data that store [x, y]. I then put that data into the CSV file. Implemented below:



The output in the CSV file was not exactly what I expected, however.



The CSV file contained all X-values, followed by all the Y-values. This is not what I wanted as I couldn’t properly put this in Excel and plot it easily. Since it looked like the matrix data makes X and Y rows, I needed to make them columns. Learned from Linear Algebra taken in a past semester, I need that taking the transpose of a row vector turns it into a column vector. MATLAB can conveniently transpose rather easy by simply using ‘ after the row you want to apply it to. Implemented below:

A close up of a black background

AI-generated content may be incorrect.

After that quick fix, our output in our CSV file is now fixed. We can clearly see which X-value corresponds with its Y-value. Here is a snippet of our CSV file:

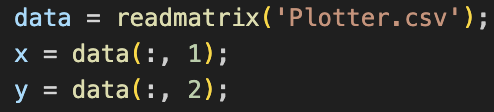
A screenshot of a computer

AI-generated content may be incorrect.

**This is the function graphed below:**

## Making The Salter

Now we need a way to salt the Y values of our function. First, we must read the data our Plotter program made. We used writematrix to write the data to our CSV, and now we are using readmatrix to read the CSV file. MATLAB has convenient methods of reading and writing files, as I learned when reading documentation. I also need to store the X and Y values into their own separate vectors. Using the colon operator that I used in my Plotter, I was able to successfully extract the data. Implemented below:



The colon inside the parentheses tells MATLAB to grab that column of our CSV file that data holds ([Source](https://www.mathworks.com/help/matlab/math/array-indexing.html)). X is stored in the first column, and Y is stored in the second.

The next step in our Salter is finding a way to generate random noise to add or subtract to our Y-values. This led me to see if I could generate random values. Using the operation **randi** allowed to create a matrix of random numbers that come from a set interval.

A black background with blue text

AI-generated content may be incorrect.

What **randi** is doing here: it is generating a random value between minVal (1) and maxVal (945), and it is generating one for every entry in y (size(y). This number is what is going to be subtracted or added to each entry in Y. This effectively allows us to obtain a random noise number for each Y-value we need to salt. This is a sample of what salt values may look like for 10 Y-values using the ranges above.

A white background with black text

AI-generated content may be incorrect.

Now I need a way to randomly subtract or add this number to each of our Y-values. Using the same logic with **randi**, I generated a column matrix of 1s and 0s for every entry of our Y-values. I did this to generate a positive or negative sign. However, 0 cannot be negative, therefore I had to figure out how to turn every 0 into a -1. MATLAB has logical indexing, and you can specifically tell it to replace any instance of a specific value with any number of your choosing ([Source](https://www.mathworks.com/help/matlab/math/array-indexing.html#f0-25789)). These two things are implemented here:

A black background with colorful letters

AI-generated content may be incorrect.

The line *randomSign(randomSign == 0) = -1;* tells MATLAB to look in the matrix randomSign and replace every 0 to a -1. Now we have successfully generated a positive or negative sign that we can apply to our noise. Using element-wise power function in MATLAB like I did for my plotter, MATLAB can calculate the sign of the noise, and add it to every value of Y, storing it into a column matrix.

A black background with blue text

AI-generated content may be incorrect.

What MATLAB is doing: let’s say Y is 150. The noise generated for that Y value is 346. That 346 is multiplied by either a 1 or -1. Then that number is added or subtracted from the Y. Example expanded below:

This is done for every single Y-value, then it is stored within a column matrix saltedY. We then simply just must get the X-values and the salted Y-values into a CSV file. We first store them into a column matrix saltedData, then we write that data to a CSV using writematrix(). Here is a snippet of our CSV file containing the salted Y-values:

A screenshot of a black screen

AI-generated content may be incorrect.

**This is the graph after the Y-values have been salted**

Now the function looks nothing like the original, which may simulate real-world data is collected originally.

### Making the Smoother

Learning about rolling averages making my smoother in Java, I tried to see if MATLAB had a convenient solution to this problem, and it did. The **movmean** function allows to take the rolling average of elements in a vector. We can determine the window which the averages are taken from using this method as well. It also handles our edge cases, like the beginning and end points of our graphs ([Source](https://www.mathworks.com/help/matlab/ref/movmean.html#bu6jxrj)).

First, just like the Salter, we read the data made by the Salter and stored the X and Y values into their respective column vectors.

A black background with text

AI-generated content may be incorrect.

Since I needed to be able to smooth the data multiple times, a for-loop is the best solution to continuously apply the movmean method to the salted Y values. User input is considered and determines how many times the for loop runs and smooth the data. Then we create a column vector that has the same salted Y values, and we continuously apply movmean to it. The window I used was 3, meaning it is a 3-point rolling average.

A black screen with text

AI-generated content may be incorrect.

The for loop will run until **i** is equal to the user input (smooths). The movmean is setup in a way that it will always smooth the previously smoothed values as well. If smoothedY inside the parentheses was replaced with just y, the for-loop would continuously only smooth original salted Y values and not replace them, effectively only smoothing them **once**. For this case, we smoothed data **20** times.

Finally, just like our Salter, we store the X values and the newly smoothed Y values and store them in a column matrix. Then using writematrix, we write our CSV file containing the new values.

A close-up of a black background

AI-generated content may be incorrect.

**Here is a snippet of the smoothed Y-values in the CSV file**

A screenshot of a computer screen

AI-generated content may be incorrect.

**This is the Graph**

You may notice something peculiar about the graph: it looks almost nothing like the original. This led me to a couple of different conclusions. My first conclusion is that my salting range is very aggressive, which is from 1 to 945. This distorts the graph so much that it is hard for the smoother to accurately smooth it back to the original. My final conclusion is that movmean may have a tendency to flatten the graph out which distorts the quadratic shape of our function, as seen towards the right of the graph.

Below is the same function plotted, salted less aggressively, and smoothed less times to prevent distortion caused by movmean. CSV snippets below:

|  |  |  |
| --- | --- | --- |
| **Plotter CSV** | **Salter CSV** | **Smoother CSV** |

**Plotter**

**Salter**

The noise range is only from 1-10, hence the changes look small on this scale

**Smoother**

The graph is nice and smooth again, showing the smoother works perfectly and can maintain the quadratic shape of the function