Dustin McDonnell

***Part 3***

Part 3 of the project has the same goals as Part 1 of the project with the difference being the inclusion of external .jar files. To facilitate this, I used Maven which has a convenient user interface with IntelliJ. I also used Choco to install Maven on my pc. The external packages I used are the Apache Commons Math Library, the Apache Commons Stats Library, and JfreeChart. I incorrectly assumed that this section of the project would be easier than coding it from scratch but my inexperience with external .jars along with that you still need to make the external methods work for what you want specifically this section ended up giving me trouble.

For the sake of keeping the different project sections self-contained there are duplicate methods from classes in part 1, I could have made the methods work for every part, but I naively believed that duplicate methods would keep it cleaner.

***Methods contained in CubeRootPart3 Class.***

***graphSpecifics()***

A duplicate method from part 1. Graph specifics gets the start point, end point, and interval of the graph from user Input then passed the information about the graph to cubeRootData().

A computer screen with many colorful lines

Description automatically generated with medium confidence

***cubeRootData()***

The goal of part 3 is to utilize external libraries and to that end I used the cube root method from the Apache math library instead of the already included Java Math class. cubeRootData() gets the ArrayList from graphSpecifics() and uses that to calculate the X and Y values.

A computer screen with text and numbers

Description automatically generated

***csvWriter()***

This creates a FileWriter object using a File path passed from createFile(), the FileWriter is then passed to Apache’s CSVprinter which then writes data in a csv format, it even has a helpful option to include headers. Apparently, there are more csv formats than just excel but for the sake of continuity that is what I used.

A computer screen shot of a code

Description automatically generated

***createFile()***

A duplicate method from part 1. This method is used to create the File object that will be passed to the csv writer. It creates a new file at the specified path if the file does not already exist.

A screenshot of a computer program

Description automatically generated

***graphData()***

This method specifies the parameters used in JfreeChart’s graph creation, titles the graph, names the axes, then saves the .jpg to the graph folder located in part 3 based on user input.

A computer screen shot of a program code

Description automatically generated

***Methods contained in File Picker Class.***

Takes a file location as its input passes that to a FileReader, which in turn gets passed to an Iterable<CSVRecord> . The values are then pulled in a loop using pre specified headers. At the time I thought this was fantastic and would make the Stock Bot portion of this project easier. How wrong I was.

A computer screen shot of a program

Description automatically generated

***filePicker()***

A duplicate method from part 1. Chooses a file based off user input in the directory to be passed to other methods in the class that alter the data. The only change is the File directory now specifies part 3 instead of part 1.

A screen shot of a computer program

Description automatically generated

***Methods contained in Manipulate Data.***

Side note I had multiple classes with duplicate names in my separate Part 1 and Part 3 folders with no issues. Then when I tried to make a duplicate class name again in the Stock Bot portion my IDE started throwing all sorts of issues with it. I genuinely do not know why. They were all in separate directories.

***saltData()***

Gets the desired salt range from user Input then salts the Y values from the ArrayList using RandomNumberGenerator from the Apache Math Library. Calls csvWriter to write the data to a new csv.

A screen shot of a computer program

Description automatically generated

***smoothData()***

I thought smoothing the data would be easier using methods from external libraries. We were even told the way to do it in lecture with “rolling mean”. How wrong I was . This method is significantly longer than the smoothing method I wrote myself in part 1 and I suspect it is also less effective. I must assume I implemented the Apache Math library incorrectly here and that it’s not supposed to be this difficult. Like before user input specifies the smoothing range which is then doubled and adds one to get the whole window range. So, if 5 were entered the loops would check 11 values. After 11 values are checked those values are averaged then added to a separate array before finally being added to another new array with the original x values.

A screen shot of a computer program

Description automatically generated

***Method contained in Part3Run.***

***run()***

Provides a simple user interface when the program is run using scanner input.

A screen shot of a computer program

Description automatically generated

***Method contained in Main class.***

***main()***

Creates a run object and calls the run method. Still not sure if main() is worth including in documentation.

A screen shot of a computer

Description automatically generated

***Part 3 Program in Action***

***Program Startup***

A screenshot of a computer

Description automatically generated

***Cube Root Data Written***

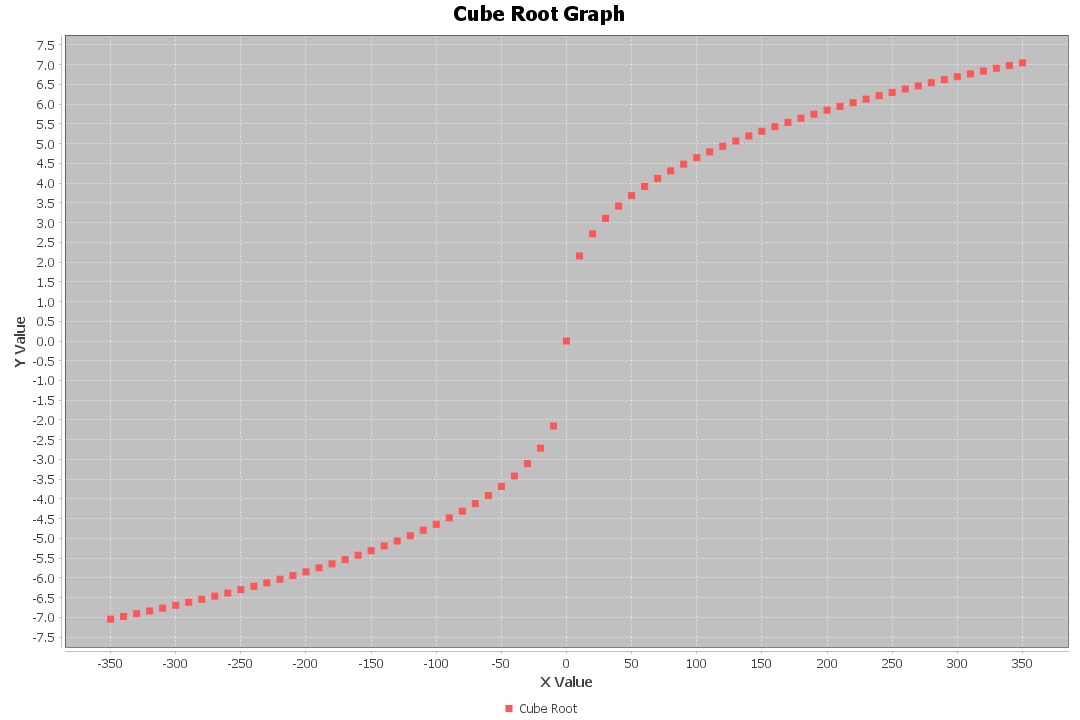
A screen shot of a computer

Description automatically generated

***Cube Root Data Graphed***

A screen shot of a computer

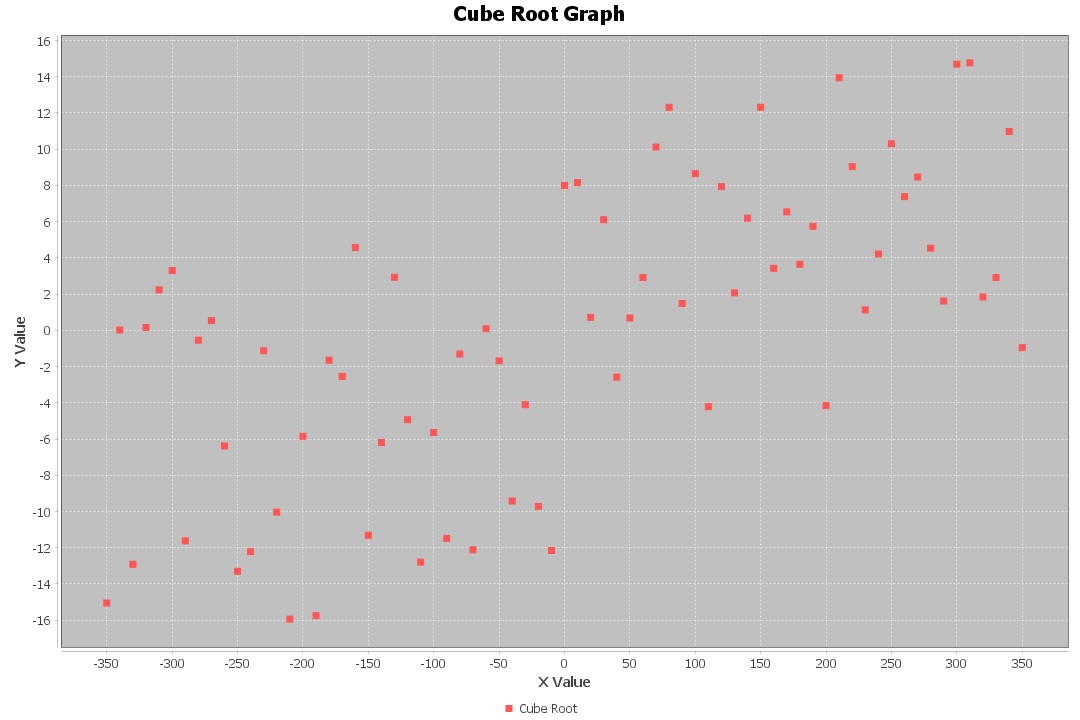
Description automatically generated



***Cube Root Data Salted and Graphed***

A computer screen with white text

Description automatically generated

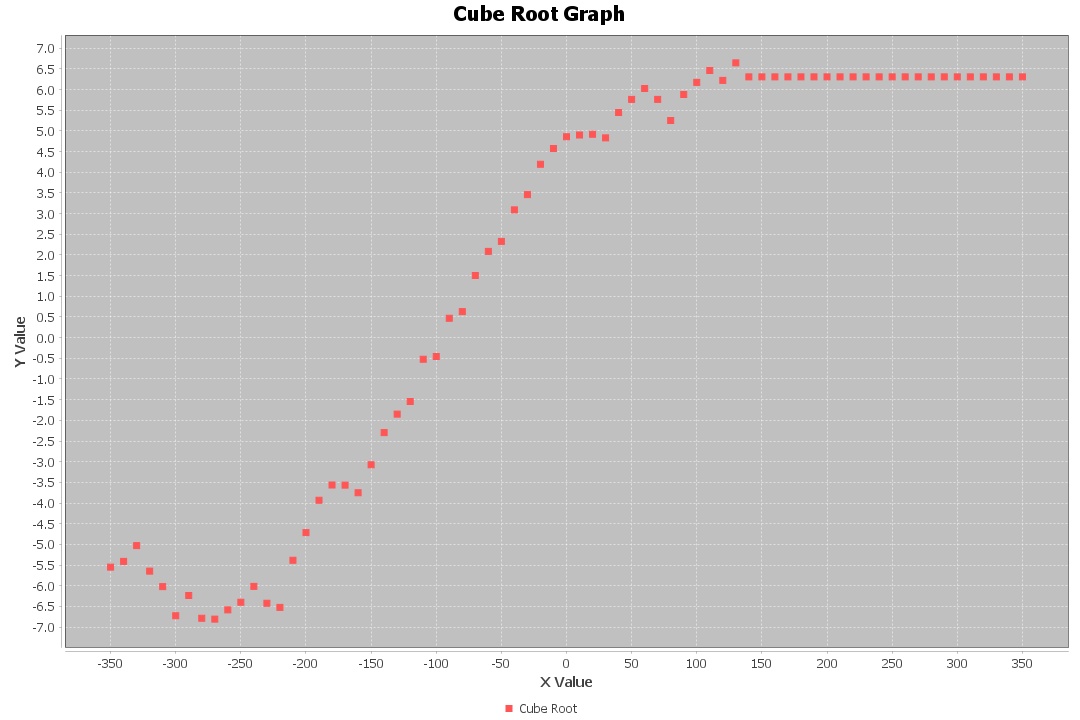


Side note, I feel like my naming conventions for files makes me look like a crazy person. There must be a cleaner and easier way to name things.

***Cube Root Data Smooth and Graphed***

A screen shot of a computer

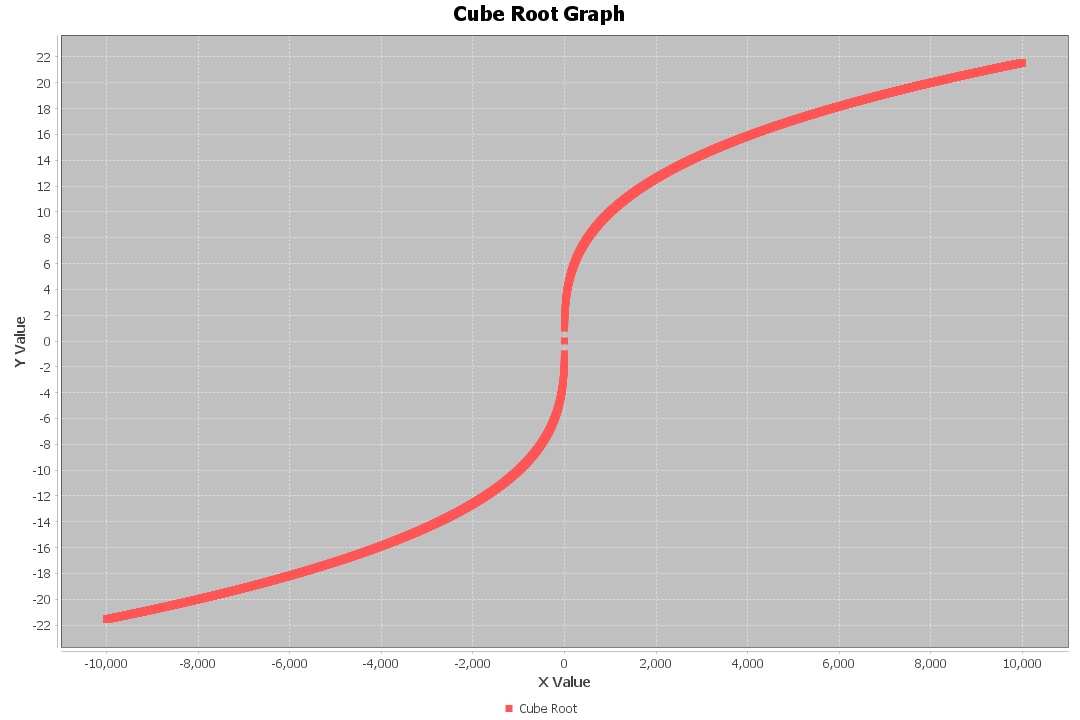
Description automatically generated



For some reason the smoothing method in part 3 takes what appears to be the last N values in the Array and sets them all to the same value. I suspected this before but it’s clear to see now with a graph that does not have that many data points. Before this I graphed an Array with significantly more points, so I did not realize this was a problem. The accuracy is off for a low amount of data points but to see if this is a significant issue I will graph another larger graph below.

***Cube Root Data Range 10000***

I have no scientific observations about the graph below but on a primal level I really enjoy looking at it.

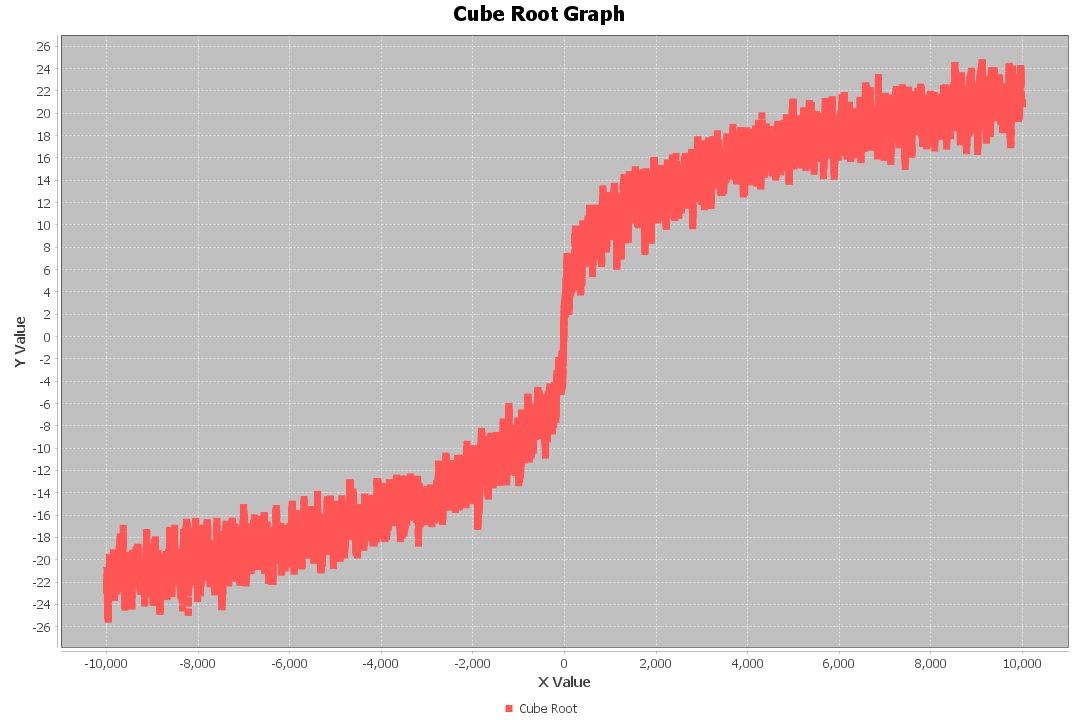


***Cube Root Data Range 10000 Salt 10***

A graph showing a curve

Description automatically generated

***Cube Root Data Range 10000 Salt 10 Smooth 10***

******

With this large number of data points accuracy does not seem to be affected by the issue that the smaller graph has, but the method from part 1 does not have the issue at any size so I would probably use my first method. Also, if I had to alter the program again, I would change the method, so the title of the graph is the same as the user input for the file instead of a static Cube Root Graph. I’d edit it myself after the fact, but it’s saved as a .jpg so barring photoshop it’s staying the way it is.