Jackson Campbell
Byron Hoy

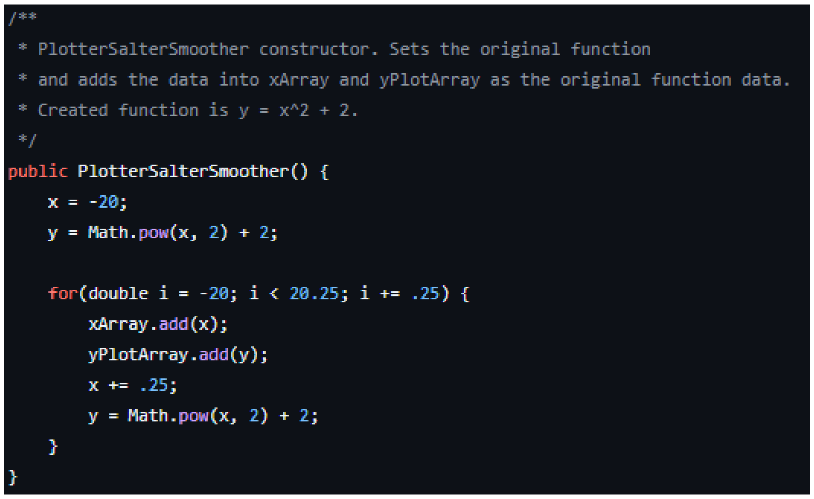
Probability and Applied Statistics
Program Documentation

PlotterSalterSmoother Project Part 1: Java

For this portion of the PlotterSalterSmoother project, all of the work was done using Java's built-in libraries and functions in order to write to CSV files and import/export data. The class itself contains set global variables:

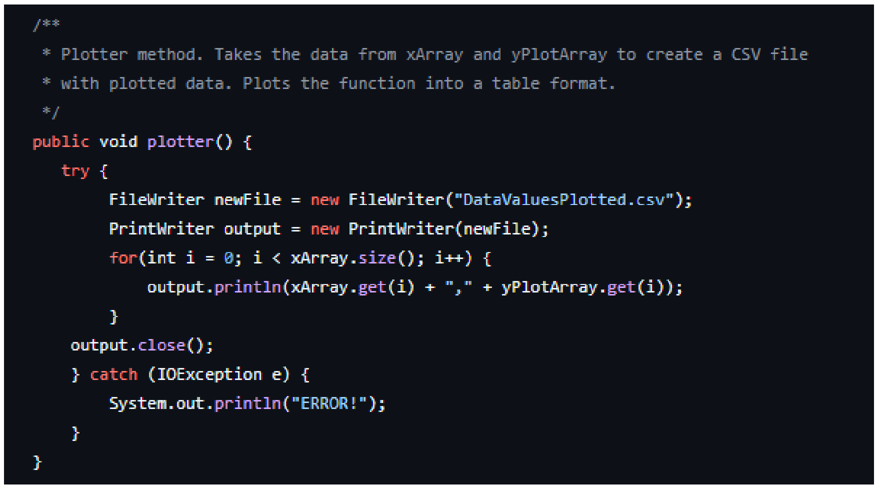
- Private double x: all of the x values in the function.
- Private double y: all of the original y values in the function.
- Private ArrayList<Double> xArray: the ArrayList containing all of the x values.
- Private ArrayList<Double> yPlotArray: the ArrayList containing all of the originally plotted y values.
- Private ArrayList<Double> ySaltArray: the ArrayList containing all of the newly salted y values from yPlotArray.
- Private ArrayList<Double> ySmoothArray: the ArrayList containing all of the newly smoothed y values from ySaltArray.

The constructor for the PlotterSalterSmoother class only initializes the original function of y = x^2 + 2, and fills in the array values for x and y from x = -20:20 with an increment of .25.



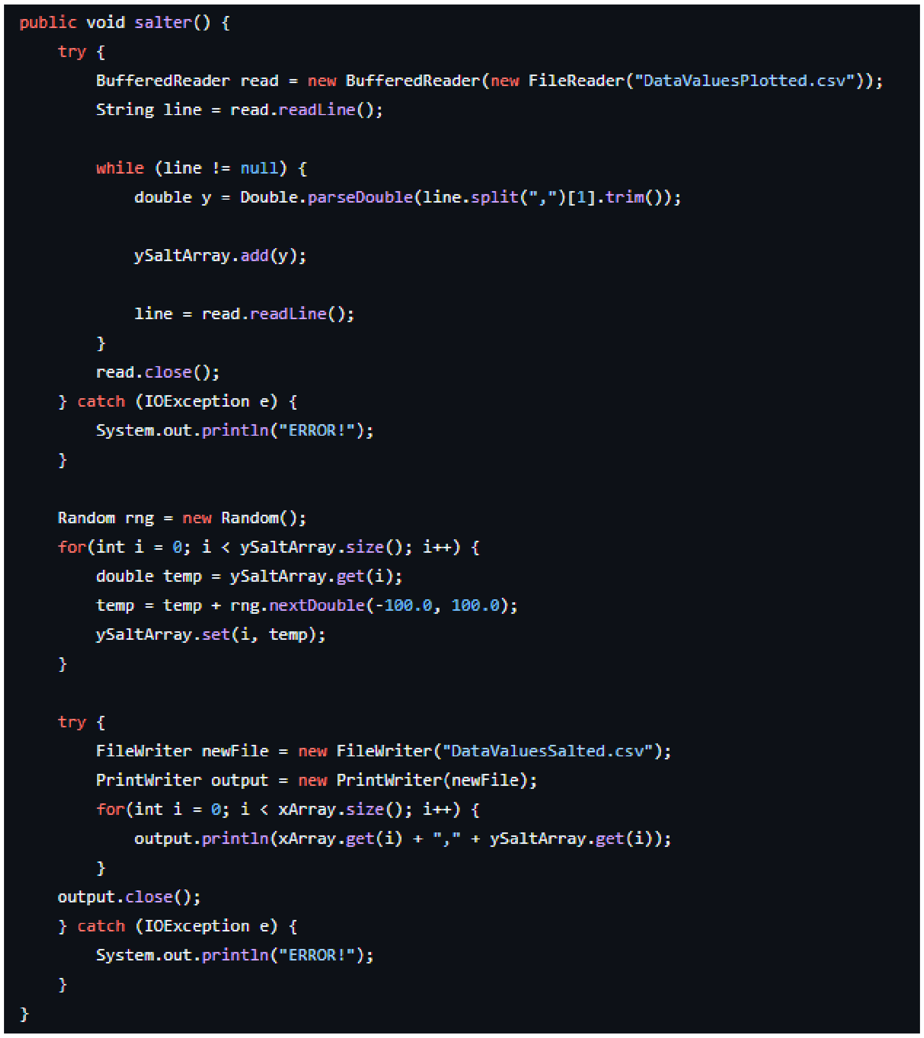
```java
/**
 * PlotterSalterSmoother constructor. Sets the original function
 * and adds the data into xArray and yPlotArray as the original function data.
 * Created function is y = x^2 + 2.
 */
public PlotterSalterSmoother() {
    x = -20;
    y = Math.pow(x, 2) + 2;

    for(double i = -20; i < 20.25; i += .25) {
        xArray.add(x);
        yPlotArray.add(y);
        x += .25;
        y = Math.pow(x, 2) + 2;
    }
}
```

The plotter method calls a try-catch block to write to a new file titled "DataValuesPlotted.csv," and contains the arrays for the originally plotted data by using the ArrayLists xArray and yPlotArray. It then closes the output, and ends the try-catch block to finish the method out.



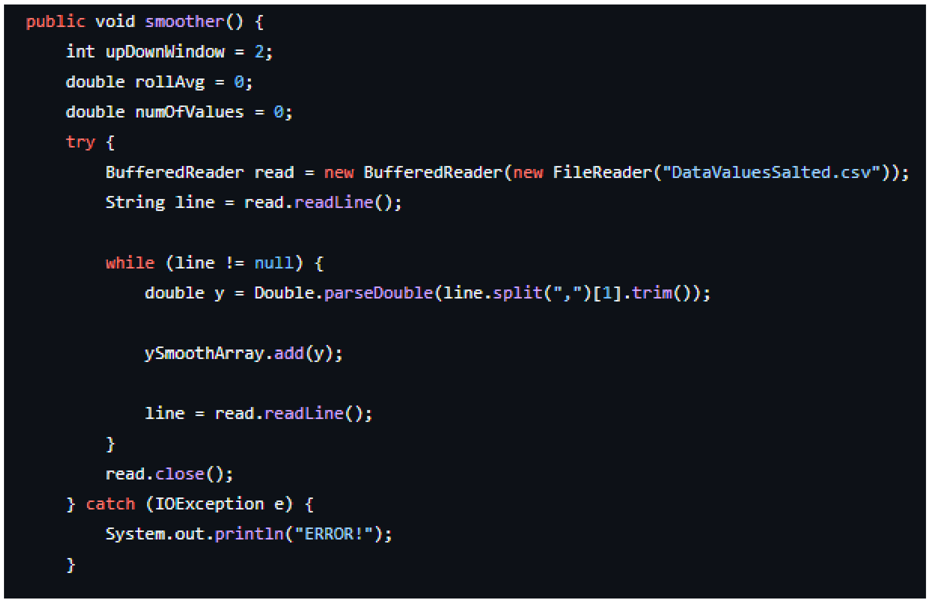
```java
/**
 * Plotter method. Takes the data from xArray and yPlotArray to create a CSV file
 * with plotted data. Plots the function into a table format.
 */
public void plotter() {
   try {
        FileWriter newFile = new FileWriter("DataValuesPlotted.csv");
        PrintWriter output = new PrintWriter(newFile);
        for(int i = 0; i < xArray.size(); i++) {
            output.println(xArray.get(i) + "," + yPlotArray.get(i));
        }
    output.close();
    } catch (IOException e) {
        System.out.println("ERROR!");
    }
}
```

The salter method also uses a try-catch block and uses the BufferedReader function from java.io to read from the previously created file in plotter(). While the current line in the csv file is not null, it parses the double value after the first comma and adds the y value to the array ySaltArray. After the first try-catch block, it then salts the data by using a Random() function from java.util to add onto each value with a random value from -100:100, and then adds that value back into the same index of the array. It then writes in another try-catch block to a new file "DataValuesSalted.csv" with the x values from xArray and y values from ySaltArray to create a second csv for use.

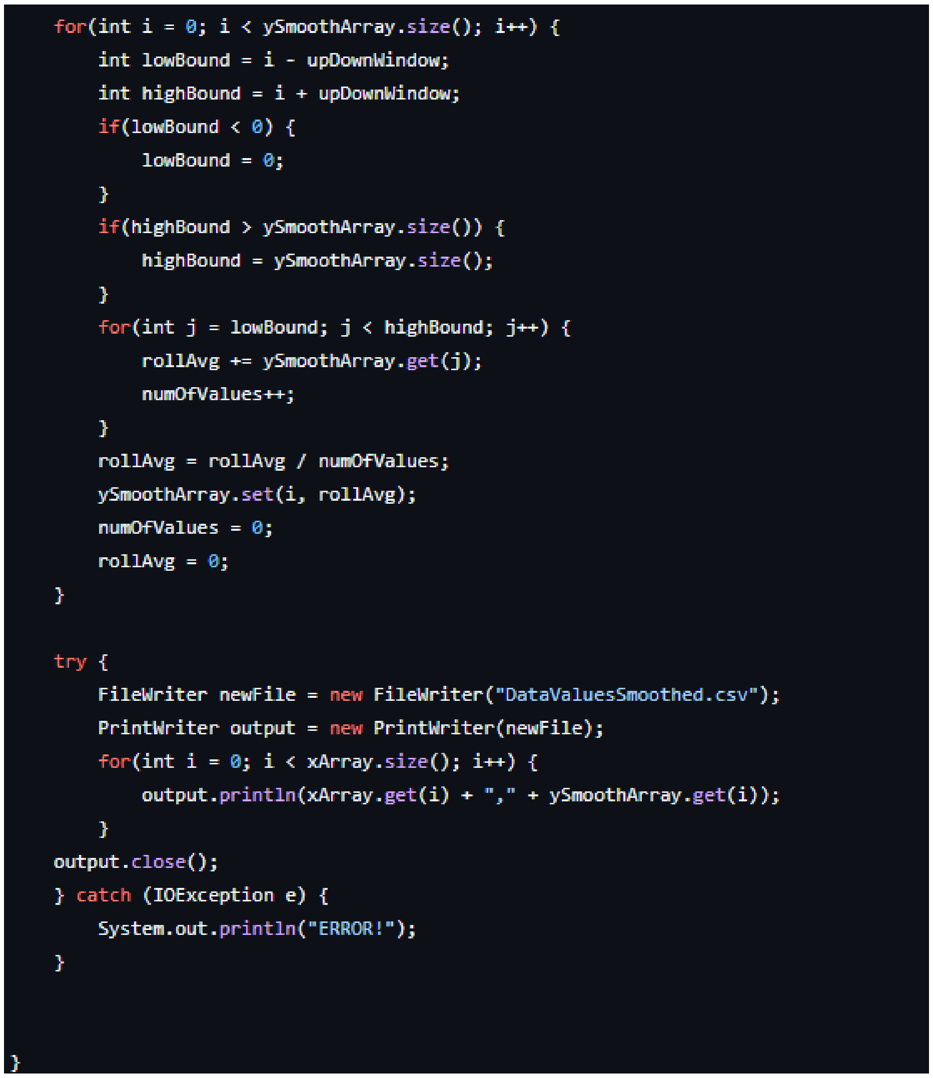


```java
public void salter() {
    try {
        BufferedReader read = new BufferedReader(new FileReader("DataValuesPlotted.csv"));
        String line = read.readLine();

        while (line != null) {
            double y = Double.parseDouble(line.split(",")[1].trim());

            ySaltArray.add(y);

            line = read.readLine();
        }
        read.close();
    } catch (IOException e) {
        System.out.println("ERROR!");
    }

    Random rng = new Random();
    for(int i = 0; i < ySaltArray.size(); i++) {
        double temp = ySaltArray.get(i);
        temp = temp + rng.nextDouble(-100.0, 100.0);
        ySaltArray.set(i, temp);
    }

    try {
        FileWriter newFile = new FileWriter("DataValuesSalted.csv");
        PrintWriter output = new PrintWriter(newFile);
        for(int i = 0; i < xArray.size(); i++) {
            output.println(xArray.get(i) + "," + ySaltArray.get(i));
        }
    output.close();
    } catch (IOException e) {
        System.out.println("ERROR!");
    }
}
```

Lastly, the smoother method creates a smoothed list of y values gathered from ySaltArray by using a rolling average. It sets a window of +- 2, and initializes the number of values and the rolling average itself with values of 0 to reset itself. It then parses the double from the salted values csv file and adds the y values to ySmoothArray. It then loops through the array, and sets a low and high bound of i – upDownWindow (which is a value of 2), and checks edge cases where lowBound is less than 0 and highBound is bigger than the array size to avoid errors. After, it adds the values to rollAvg and increases the count in numOfValues, which were both originally set to 0 to ensure they work properly. After looping through the low bound and high bound's worth of values to create the average number, it then properly sets the average by calculating rollAvg/numOfValues to grab the average for that specific set of values, and then sets it to the current index in ySmoothArray. It then resets rollAvg and numOfValues for the next iteration of the loop. Afterwards, it writes the data from xArray and ySmoothArray to create the last csv, "DataValuesSmoothed.csv."

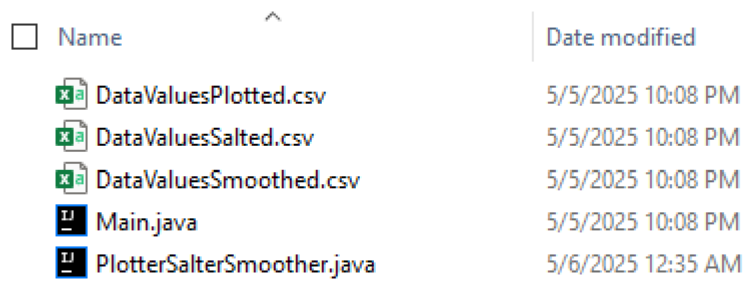


```java
public void smoother() {
    int upDownWindow = 2;
    double rollAvg = 0;
    double numOfValues = 0;
    try {
        BufferedReader read = new BufferedReader(new FileReader("DataValuesSalted.csv"));
        String line = read.readLine();

        while (line != null) {
            double y = Double.parseDouble(line.split(",")[1].trim());

            ySmoothArray.add(y);

            line = read.readLine();
        }
        read.close();
    } catch (IOException e) {
        System.out.println("ERROR!");
    }
```



```java
    for(int i = 0; i < ySmoothArray.size(); i++) {
        int lowBound = i - upDownWindow;
        int highBound = i + upDownWindow;
        if(lowBound < 0) {
            lowBound = 0;
        }
        if(highBound > ySmoothArray.size()) {
            highBound = ySmoothArray.size();
        }
        for(int j = lowBound; j < highBound; j++) {
            rollAvg += ySmoothArray.get(j);
            numOfValues++;
        }
        rollAvg = rollAvg / numOfValues;
        ySmoothArray.set(i, rollAvg);
        numOfValues = 0;
        rollAvg = 0;
    }

    try {
        FileWriter newFile = new FileWriter("DataValuesSmoothed.csv");
        PrintWriter output = new PrintWriter(newFile);
        for(int i = 0; i < xArray.size(); i++) {
            output.println(xArray.get(i) + "," + ySmoothArray.get(i));
        }
    output.close();
    } catch (IOException e) {
        System.out.println("ERROR!");
    }


}
```

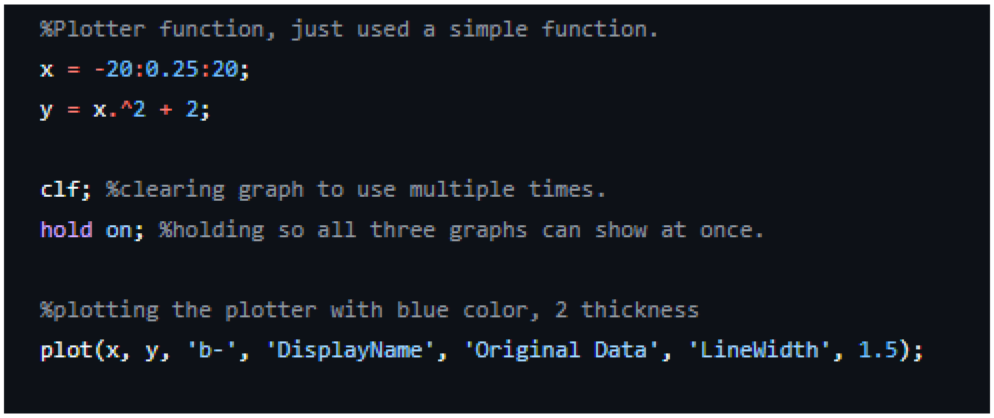
A run method is also included to run all three functions, thus creating the given csv files above. A screenshot of the file directory is included below to show the program works properly:



Name	Date modified
DataValuesPlotted.csv	5/5/2025 10:08 PM
DataValuesSalted.csv	5/5/2025 10:08 PM
DataValuesSmoothed.csv	5/5/2025 10:08 PM
Main.java	5/5/2025 10:08 PM
PlotterSalterSmoother.java	5/6/2025 12:35 AM

PlotterSalterSmoother Project Part 2: MatLab

Next, we were tasked with learning how to use MatLab, or in our case Octave (which is a free version of MatLab) to create these same functions in part 1. MatLab is a math-oriented language, and contains functions to make creating this data much easier. After looking at short tutorials and messing around with the program myself, I was able to create a plotted graph using the same function above. To plot all three graphs, I had to use the “hold” function to let Octave know I wanted to write to the graph still. It was then easy to plot my range of data for the first graph as I simply used the plot(x, y) function to plot the original function with the color blue using parameter ‘b-‘, the display name “Original Data” by using syntax (‘DisplayName’, ‘Original Data’), and then setting the line width to 1.5 by using similar syntax to DisplayName, but using ‘LineWidth’ instead along with the width value itself.



```
%Plotter function, just used a simple function.
x = -20:0.25:20;
y = x.^2 + 2;

clf; %clearing graph to use multiple times.
hold on; %holding so all three graphs can show at once.

%plotting the plotter with blue color, 2 thickness
plot(x, y, 'b-', 'DisplayName', 'Original Data', 'LineWidth', 1.5);
```

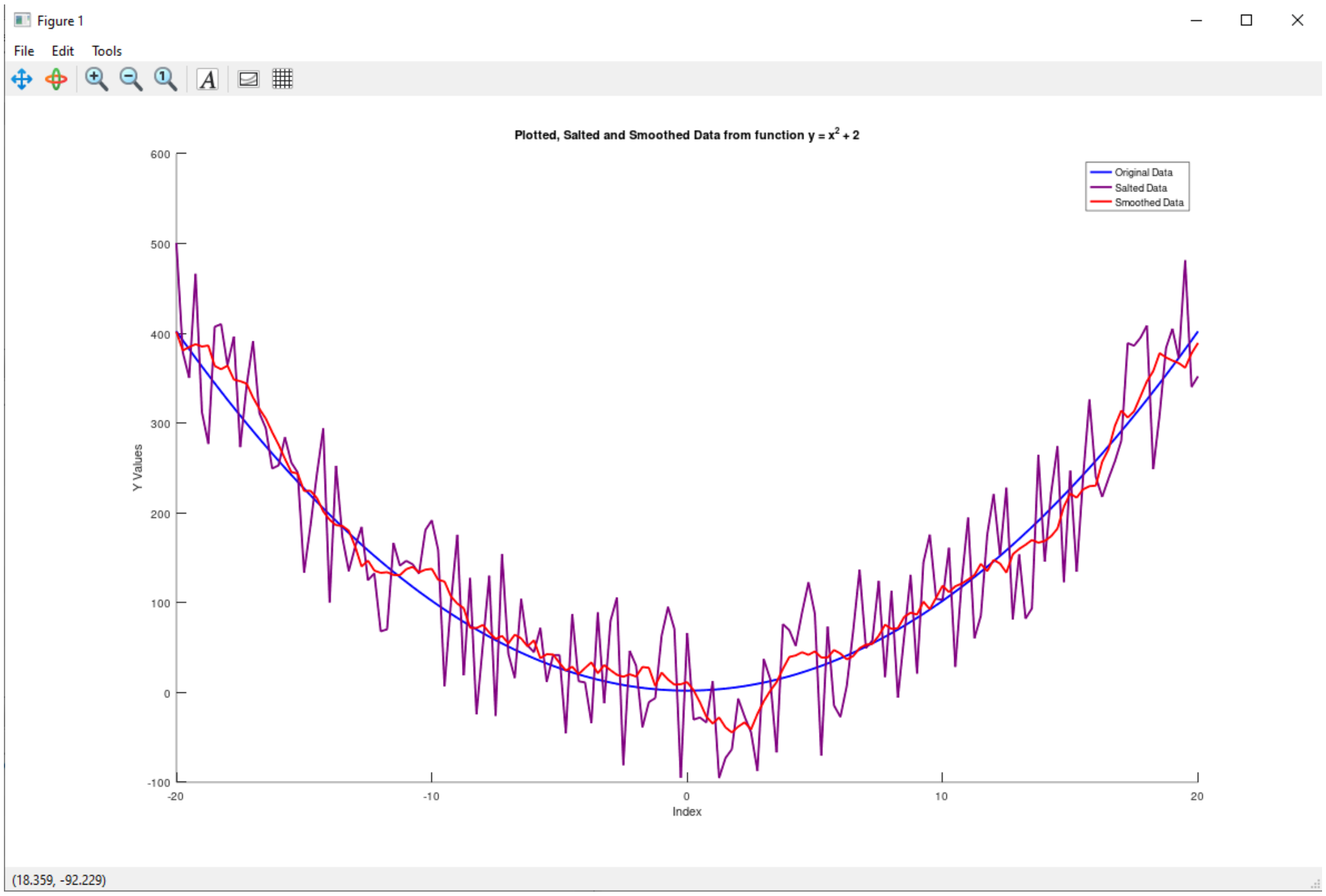
After that first line was plotted, I then looped through the values of y and created a temp value, used the built in rand() function, which gives a random decimal value from 0 to 1, took that value, multiplied it by 200, and then subtracted 100 from it to create the random value to add to each index of the salted data. To finish, I simply ended the for loop with endfor to signify the loop was over.

```
%loop to salt the data comparatively to the java file.
for i = 1:length(y)
  temp = y(i);
  temp = temp + (rand() * 200 - 100);
  y(i) = temp;
endfor;
```

I then set the window size to a much larger size of 10, and used the built in function of movMean(y, winSize) to create a rolling average for the y values along with a set window size for the average. Once I set that average to an array "smoother" I could plot the two other functions using the same as above but with the color purple for the salted data, and the color red for the smoothed data. I then added a legend so the display names could be seen, and then added a legend with a title "Plotted, Salted and Smoothed Data from function y = x^2 + 2," and labeled X as "Index" and Y as "Y Values." I then close the hold, as I no longer need to add data to the given graph, and the program is then over.

```
%setting the window size and smoothing with a moving average.
winSize = 10;
smoother = movmean(y, winSize);

%plotting the salted and smoothed graphs.
%salted is purple with thickness 2.
%smoothed is red with thickness 2.
plot(x, y, 'Color', '#800080', 'LineWidth', 1.5, 'DisplayName', 'Salted Data');
plot(x, smoother, 'r-', 'LineWidth', 1.5, 'DisplayName', 'Smoothed Data');
legend;
title('Plotted, Salted and Smoothed Data from function y = x^2 + 2');
xlabel('Index');
ylabel('Y Values');
hold off; %stop the hold on graph display.
```

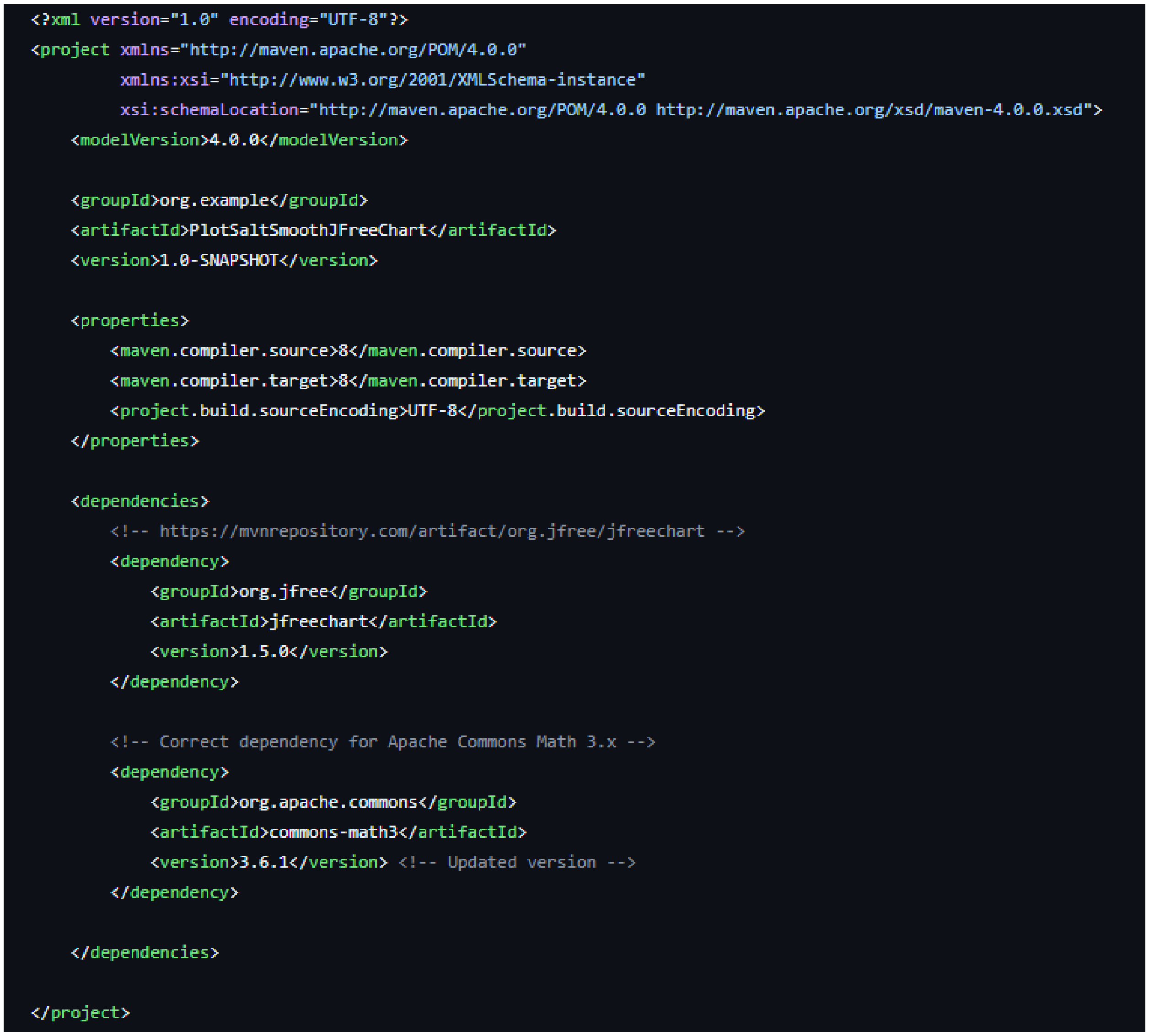
After this was done, I could run my functions to create a graph as displayed below with all of the given data:



After all was said and done, Octave was a really easy program to get ahold of and use and made creating this much easier than before.

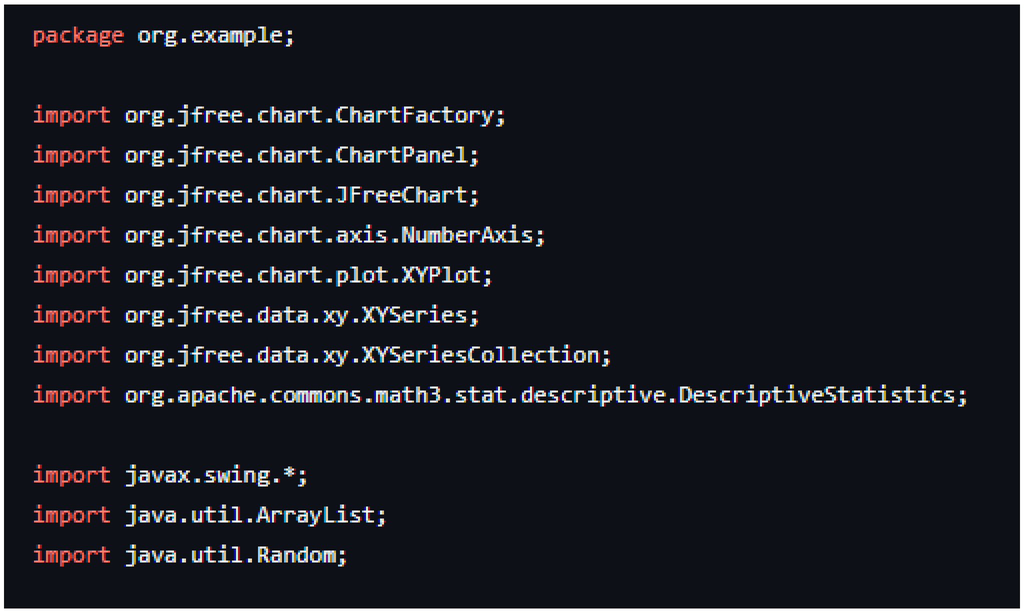
PlotterSalterSmoother Project Part 3: JFreeChart and Apache Commons

As a final part to PlotterSalterSmoother, we were tasked with using Maven to implement JFreeChart and Apache Commons to handle graph creation and rolling average functions. For starters, I set the XML file up using help from Jessica Smith from last semester to properly set the dependencies for both JFreeChart and Apache Commons:



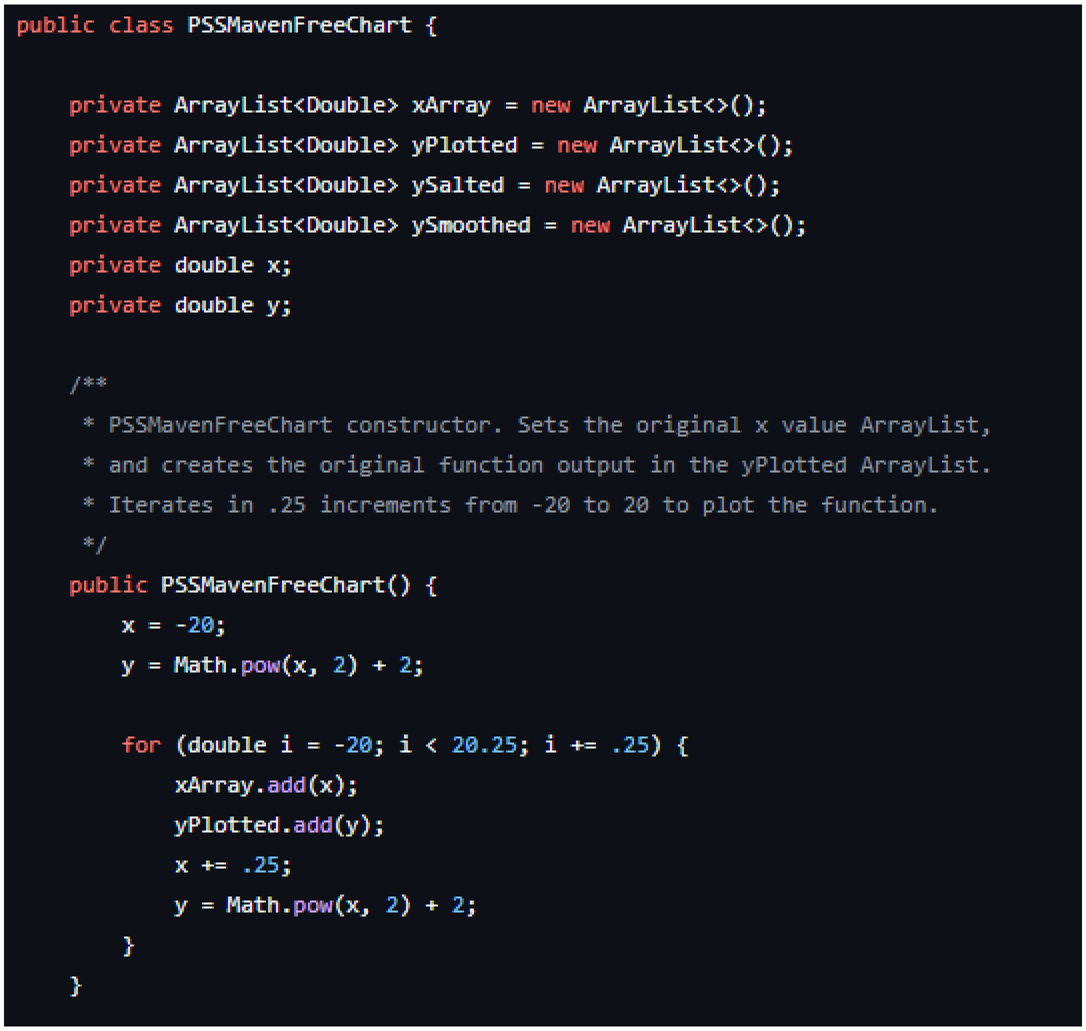
```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>PlotSaltSmoothJFreeChart</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.jfree/jfreechart -->
        <dependency>
            <groupId>org.jfree</groupId>
            <artifactId>jfreechart</artifactId>
            <version>1.5.0</version>
        </dependency>

        <!-- Correct dependency for Apache Commons Math 3.x -->
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-math3</artifactId>
            <version>3.6.1</version> <!-- Updated version -->
        </dependency>

    </dependencies>

</project>
```

After this, I was able to import the libraries from JFreeChart and Apache Commons in order to properly use them. Credit again to Jessica Smith in setting all of this up:



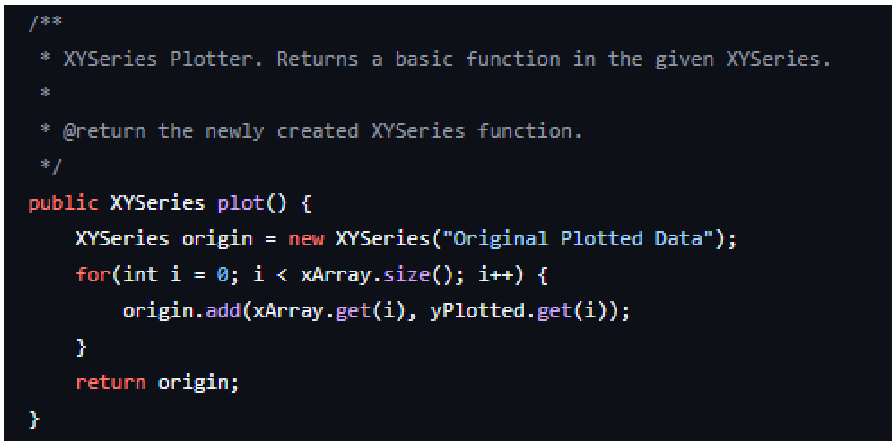
```java
package org.example;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.plot.XYPlot;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;
import org.apache.commons.math3.stat.descriptive.DescriptiveStatistics;

import javax.swing.*;
import java.util.ArrayList;
import java.util.Random;
```

Just like part 1, the same ArrayLists are also used, along with the double values for x and y. The same constructor setup is also used to fill the original xArray and yPlotted array.



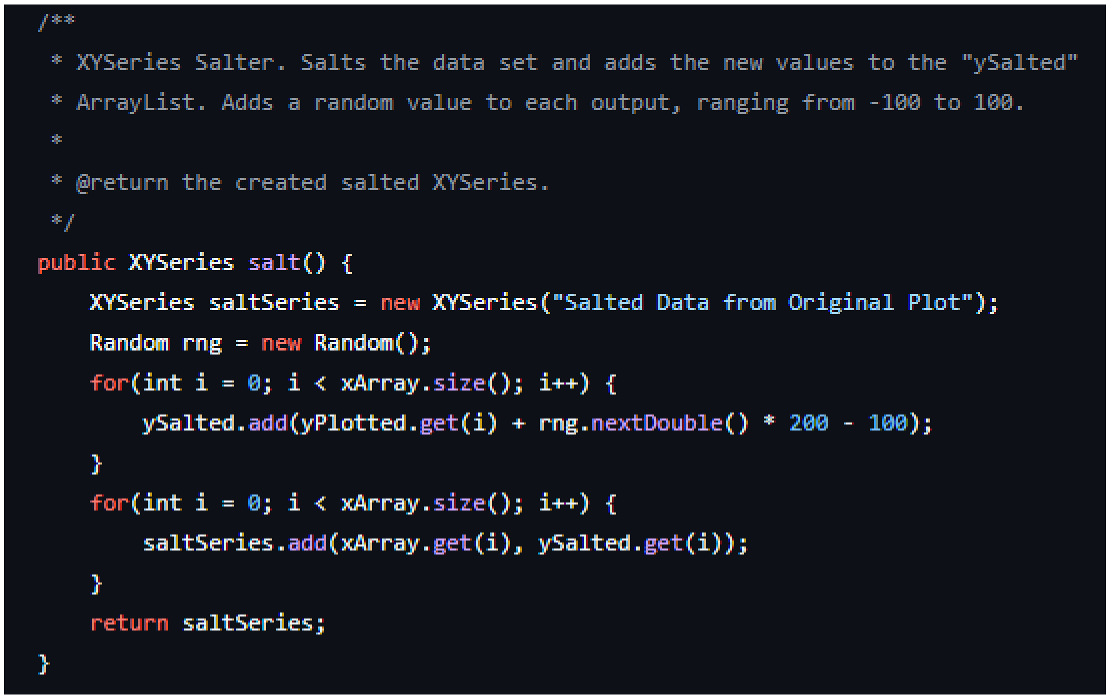
```java
public class PSSMavenFreeChart {

    private ArrayList<Double> xArray = new ArrayList<>();
    private ArrayList<Double> yPlotted = new ArrayList<>();
    private ArrayList<Double> ySalted = new ArrayList<>();
    private ArrayList<Double> ySmoothed = new ArrayList<>();
    private double x;
    private double y;

    /**
     * PSSMavenFreeChart constructor. Sets the original x value ArrayList,
     * and creates the original function output in the yPlotted ArrayList.
     * Iterates in .25 increments from -20 to 20 to plot the function.
     */
    public PSSMavenFreeChart() {
        x = -20;
        y = Math.pow(x, 2) + 2;

        for (double i = -20; i < 20.25; i += .25) {
            xArray.add(x);
            yPlotted.add(y);
            x += .25;
            y = Math.pow(x, 2) + 2;
        }
    }
```

Different to part 1 though, is how the plot() method is created, I instead create a new XYSeries which is used to create a new graph image later on in the allThree() method. I give the XYSeries the title “Original Plotted Data” and fill it with x and y values from xArray and yPlotted. I then return origin, the name of the XYSeries to finish this first method.



```java
/**
 * XYSeries Plotter. Returns a basic function in the given XYSeries.
 *
 * @return the newly created XYSeries function.
 */
public XYSeries plot() {
    XYSeries origin = new XYSeries("Original Plotted Data");
    for(int i = 0; i < xArray.size(); i++) {
        origin.add(xArray.get(i), yPlotted.get(i));
    }
    return origin;
}
```

I then create another XYSeries, saltSeries and give it the title “Salted Data from Original Plot.” I then use Java’s Random class to salt the data similar to how I salted it in Octave by adding (rng.nextDouble() * 200 – 100) to the current value in yPlotted, and then proceeding to add that value to the ySalted array. I then do the same as the plot() method and iterate through and add the xArray and ySalted data to saltSeries to create the second data set for the graph.



```java
/**
 * XYSeries Salter. Salts the data set and adds the new values to the "ySalted"
 * ArrayList. Adds a random value to each output, ranging from -100 to 100.
 *
 * @return the created salted XYSeries.
 */
public XYSeries salt() {
    XYSeries saltSeries = new XYSeries("Salted Data from Original Plot");
    Random rng = new Random();
    for(int i = 0; i < xArray.size(); i++) {
        ySalted.add(yPlotted.get(i) + rng.nextDouble() * 200 - 100);
    }
    for(int i = 0; i < xArray.size(); i++) {
        saltSeries.add(xArray.get(i), ySalted.get(i));
    }
    return saltSeries;
}
```

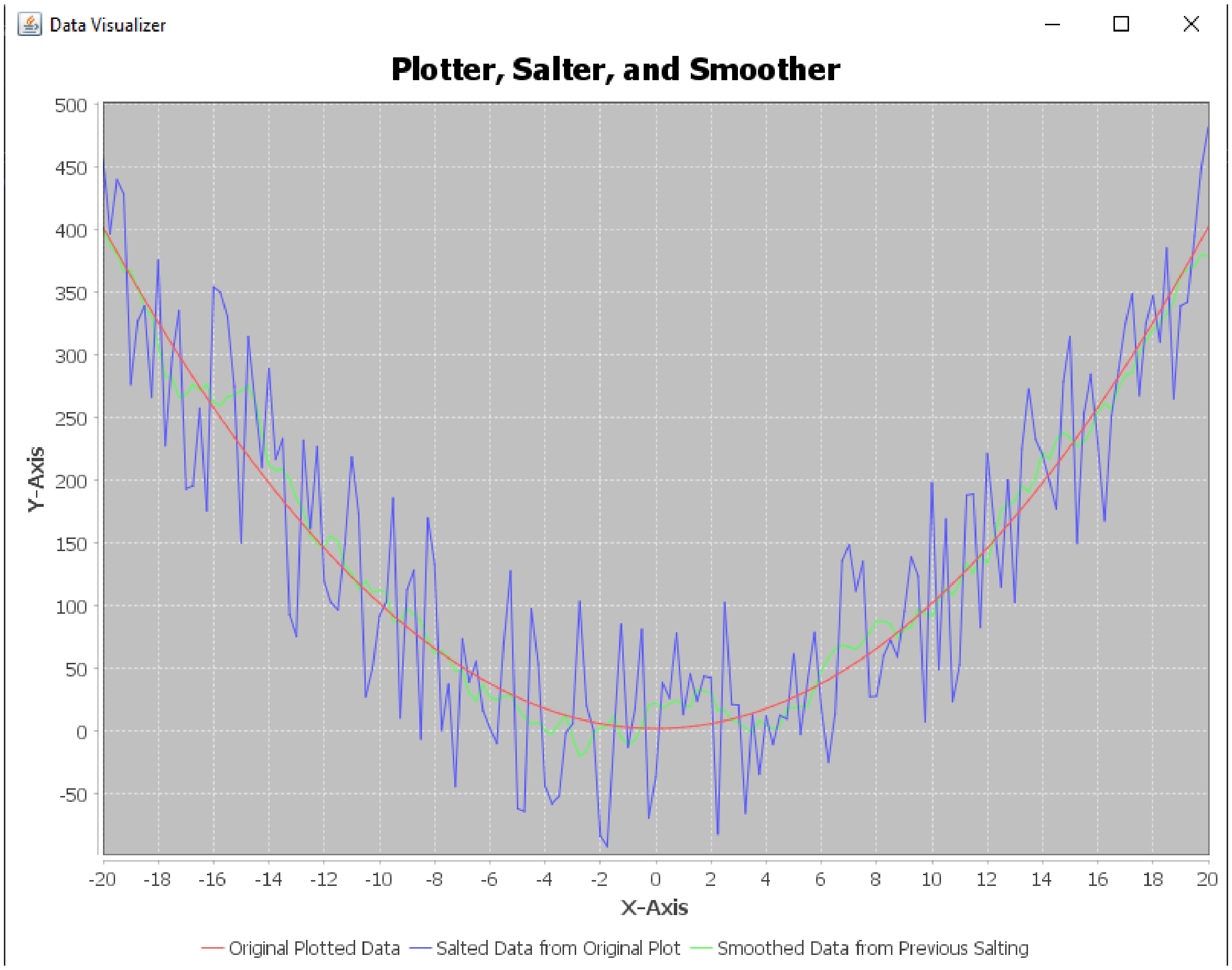
I then created the last XYSeries, smoothSeries and give it the title “Smoothed Data from Previous Salted” to signify these values are smoothed. I also attempt to use Apache Commons’s DescriptiveStatistics to create a rolling average, but end up failing with my bounds later on when the graph prints. When setting up DescriptiveStatistics, I set a window size of 5, and an original count value of 0. I then loop through ySalted and add the value to ds, and if the count of values exceeds the window size, I then add the rolling average value to ySmoothed. I believe the way I handled adding the rolling average values to ySmoothed is what caused my bounds to be improper, as it does not cover all the way to 20. Otherwise, setting up the rolling average is exactly the same as part one of this project portion as shown in the code below:

```java
public XYSeries smooth() {
    DescriptiveStatistics ds = new DescriptiveStatistics();
    XYSeries smoothSeries = new XYSeries("Smoothed Data from Previous Salting");
    int window = 5;
    int count = 0;
    ds.setWindowSize(window);

    for(Double y : ySalted) {
        ds.addValue(y);
        if(count >= window) {
            ySmoothed.add(ds.getMean());
        }
        count++;
    }

    /*
    for(int i = 0; i < ySalted.size(); i++) {
        double rollAvg = 0;
        double numOfValues = 0;
        int lowBound = i - window;
        int highBound = i + window;
        if(lowBound < 0) {
            lowBound = 0;
        }
        if(highBound > ySalted.size()) {
            highBound = ySalted.size();
        }
        for(int j = lowBound; j < highBound; j++) {
            rollAvg += ySalted.get(j);
            numOfValues++;
        }
        double temp = rollAvg / numOfValues;
        ySmoothed.add(temp);
    }
    */
```

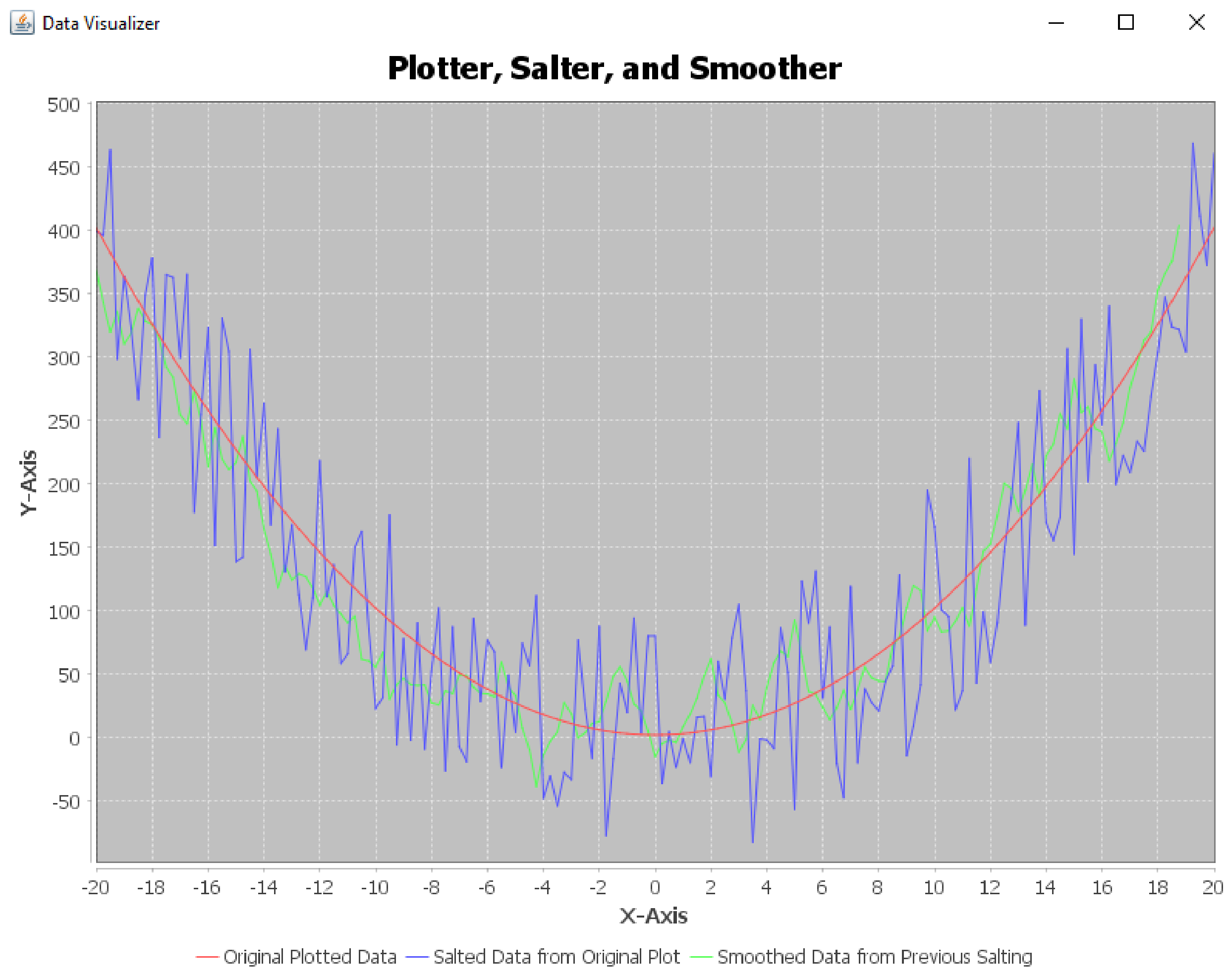
The rest of this method is just adding the values to the XYSeries to create a proper output.

Lastly, the allThree() method is entirely different to part one, as a graph must now be created using JFreeChart. I created a new XYSeriesCollection called dataSet, to which I add all three of the previously created series. I then used JFreeChart's ChartFactory.createXYLineChart to make a new graph and created a title "Plotter, Salter, and Smoother" for the graph, and gave simple X/Y-Axis titles to each axis. Then, a new XYPlot plot was created and set to chart.getXYPlot to grab the plotter itself. I then plotted the x and y axis on the graph with ranges -20:20 and -98:20^2 + 2 + 100 respectively. I then used ChartPanel to create an actual window for the chart, with a resolution of 800x600 originally. Once this was all set, I was then able to make a new JFrame and set the close operation to exit on closing, and added the chartPanel created above it, and set it to visible to display the graph. Finally, I used SwingUtilities.invokeLater to run allThree(), therefore creating the charts below:

Using the original rolling average calculation:



Using Apache's DescriptiveStatistics:



In this graph, you can see that the green line doesn't extend all the way to 20, signifying that I did not get Apache working properly. The original code is left in, and is simply commented out.