PSS1 Report

Mouaz Ali

Table of Contents

Plotter
The "Plotter" class is designed to create a CSV (Comma-Separated Values) file, or to plot data.

To make this happen, the Plotter uses a FileWriter, a tool that writes data into files. Specifically, it writes data into a file with a ".csv" extension, which is a format commonly used for storing table-like data. The FileWriter does all the heavy lifting, creating the file if it doesn't exist and adding data to it line by line.

Each line in the CSV file represents a point on the graph, with the x and y coordinates separated by commas. This makes it easy to read and share the data, especially if you want to use it in spreadsheet programs like Excel, making the Plotter class a convenient way to generate and export graph data efficiently.

Here's a description of what the “Plotter” class does:

1. File Creation: When you call the plot method, the class creates a new CSV file to store the plotted data. The filename is specified by the `fileName` parameter, with ".csv" added to the end.
```
// Create CSV file
File plotted = new File(fileName + ".csv");
```
2. Header: The class adds a header to the CSV file, specifying the names of the columns. By default, the header includes "x" and "y" to represent the coordinates, along with the equation of the quadratic function used to generate the data.
```
// Header for the CSV file
String header = "x, y, y = x^2 - 4x + 3";
```
3. Data Generation: For each x-coordinate within the specified range (`min` to `max`), the class calculates the corresponding y-coordinate using the quadratic function $y = x^2 - 4x + 3$. In the test class, you will see that this equation is applied to each counter value within the specified range (min to max) and incremented by the interval.
```
// Calculate the corresponding y value using the quadratic function
double y = Math.pow(counter, 2) - 4 * counter + 3;
```
4. Rounding: Before writing the coordinates to the CSV file, both the x and y values are rounded to the tenths place to ensure consistency and readability.
```
// Round the x and y values to the tenth place
String formattedX = String.format("%.1f", counter);
String formattedY = String.format("%.1f", y);
```
5. Data Writing: It writes the calculated coordinates to the CSV file. Each line in the file represents a point on the graph, with the x-coordinate and corresponding y-coordinate separated by a comma.
```
// Write the formatted x and y values to the CSV file
fileWriter.write(formattedX + "," + formattedY);
```
6. File Closing: Finally, the FileWriter is closed to ensure that all data is written to the file..
```
// Close the FileWriter
fileWriter.close();
```

Full Plotter Class Code:

```java
import java.io.*;

public class Plotter {
    // Method to plot data and save it to a CSV file
    public void plot(String fileName, int min, int max, double interval) {
        // Create CSV file
        File plotted = new File(fileName + ".csv");

        // Header for the CSV file
        String header = "x, y, y = x^2 - 4x + 3";

        try {
            // FileWriter object to write to the CSV file
            FileWriter fileWriter = new FileWriter(plotted);

            // Header to the CSV file
            fileWriter.write(header);
            // Move to the next line
            fileWriter.write(System.lineSeparator());

            // Loop through the specified range of x values
            for (double counter = min; counter < max; counter += interval) {
                // Calculate the corresponding y value using the quadratic function
                double y = Math.pow(counter, 2) - 4 * counter + 3;

                // Round the x and y values to the tenth place
                String formattedX = String.format("%.1f", counter);
                String formattedY = String.format("%.1f", y);

                // Write the formatted x and y values to the CSV file
                fileWriter.write(formattedX + "," + formattedY);
                // Move to the next line
                fileWriter.write(System.lineSeparator());
            }
            // Close the FileWriter
            fileWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Salter

The Salter class is designed to apply random salt values to the y-coordinates of points in a CSV file containing plotted data. This process introduces variability or noise into the data, simulating real-world scenarios where measurements may not be perfectly accurate. In class, we had discussed how Apple search salts data to protect user privacy.

Some of the functionalities of the salter class include:

1. File Reading: The salter reads the input CSV file containing plotted data, extracting the x and y coordinates from each line and storing them in an ArrayList.

```java
// Read the input CSV file containing plotted data
FileReader fileReader = new FileReader(fileName);
BufferedReader bufferedReader = new BufferedReader(fileReader);
```

2. Salt Generation: Random salt values are generated within the specified range (saltRangeMin to saltRangeMax) using the Random class.

```
// Generate a random salt value within the specified range
double saltValue = saltRangeMin + (saltRangeMax - saltRangeMin) * rand.nextDouble();
```

3. Application: For each y-coordinate in the data, a randomly generated decision determines whether to add or subtract the salt value, introducing variation.

```
if (decision)
    temp += saltValue;
else
    temp -= saltValue;
```

4. File Writing: The salted data is written to a new CSV file named "SaltedValues.csv" using a FileWriter. The difference in this file is that the y values are changed.

```
FileWriter fileWriter = new FileWriter(salted);
```

5. File Closing: FileWriter is closed after all data is written to the file.

```
fileWriter.close();
```

Full salter class code:

```
public class Salter {
    public void salt(String fileName, int saltRangeMin, int saltRangeMax) {
        // ArrayList to store x and y values
        ArrayList<String> xyValues = new ArrayList<>();
        File salted = new File("SaltedValues.csv");
        // For generating random values
        Random rand = new Random();
        try {
            // Read the input CSV file containing plotted data
            FileReader fileReader = new FileReader(fileName);
            BufferedReader bufferedReader = new BufferedReader(fileReader);
            String nextLine;
            int count = 0;
            // Reads lines of input file
            for (nextLine = bufferedReader.readLine(); nextLine != null; nextLine = bufferedReader.readLine(), count++) {
                // Skips header
                if (count > 0) {
                // Split each line into x and y values and add them to the ArrayList
                    String[] lineValues = nextLine.split(",");
                    xyValues.addAll(Arrays.asList(lineValues));
                }
            }
            bufferedReader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        // Iterate through the ArrayList to apply salt to y values
        for (int i = 0; i < xyValues.size(); i++) {
            if (i % 2 == 1) {
                // Generate a random salt value within the specified range
                double saltValue = saltRangeMin + (saltRangeMax - saltRangeMin) * rand.nextDouble();
                // Random decision to determine whether to add or subtract salt
                boolean decision = rand.nextBoolean();
                Double temp = Double.parseDouble(xyValues.get(i));
                // Apply salt based on the random decision
                if (decision)
                    temp += saltValue;
                else
                    temp -= saltValue;
                // Update the ArrayList with the salted y value
                xyValues.set(i, temp.toString());
            }
        }
        try {
            // Write the salted data to the output CSV file
            FileWriter fileWriter = new FileWriter(salted);
            for (int i = 0; i < xyValues.size(); i++) {
                fileWriter.write(xyValues.get(i) + ",");
                if (i % 2 != 0)
                    fileWriter.write(System.lineSeparator());
            }
            fileWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Smoother

The Smoother class implements a moving average smoothing algorithm to smooth out the y-values of the dataset read from a salted CSV file. The goal of this algorithm is to reduce noise in the dataset by replacing each y-value with a smoothed average value computed from its neighboring values, resulting in a smoother representation of the data.

During the smoothing process, this class sums up the y-values within the window and divides the sum by the number of values to compute the average. The size of the window, in this case, is determined by the windowValue parameter.

Here is a breakdown of the components in this class:

1. Reading Input File: This code reads each line of the input saltedValues CSV file, splits it by comma to extract the x and y values, and adds them to separate ArrayLists.

```
FileReader fileReader = new FileReader(input);
BufferedReader bufferedReader = new BufferedReader(fileReader);
String nextLine;
// Read and parse the input CSV file
for (nextLine = bufferedReader.readLine(); nextLine != null; nextLine = bufferedReader.readLine()) {
    String[] lineValues = nextLine.split(",");
    xValues.add(Double.parseDouble(lineValues[0])); // Add x-value to list
    yValues.add(Double.parseDouble(lineValues[1])); // Add y-value to list
}
```

2. Smoothing: The movingAverage method calculates the smoothed y-values using a moving average algorithm with a specified window value.

```
// Smooth the y-values using moving average
ArrayList<Double> smoothedYValues = movingAverage(yValues, windowValue);
```

3. Writing to Output File: After smoothing the y-values, the code writes the original x-values and the smoothed y-values to the output CSV file.

```
FileWriter fileWriter = new FileWriter(output);
// Write x-values and corresponding smoothed y-values to the output CSV file
for (int i = 0; i < xValues.size(); i++) {
    fileWriter.write(xValues.get(i) + "," + smoothedYValues.get(i));
    fileWriter.write(System.lineSeparator());
}
```

Full smoother class code:

```
public class Smoother {
    public void smooth(String inputFileName, String outputFileName, int windowValue) {
        ArrayList<Double> xValues = new ArrayList<>(); // Store x-values
        ArrayList<Double> yValues = new ArrayList<>(); // Store y-values
        File input = new File(inputFileName);
        File output = new File(outputFileName);

        try {
            FileReader fileReader = new FileReader(input);
            BufferedReader bufferedReader = new BufferedReader(fileReader);
            String nextLine;
            // Read and parse the input CSV file
            for (nextLine = bufferedReader.readLine(); nextLine != null; nextLine = bufferedReader.readLine()) {
                String[] lineValues = nextLine.split(",");
                xValues.add(Double.parseDouble(lineValues[0])); // Add x-value to list
                yValues.add(Double.parseDouble(lineValues[1])); // Add y-value to list
            }
            bufferedReader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Smooth the y-values using moving average
        ArrayList<Double> smoothedYValues = movingAverage(yValues, windowValue);

        try {
            FileWriter fileWriter = new FileWriter(output);
            // Write x-values and corresponding smoothed y-values to the output CSV file
            for (int i = 0; i < xValues.size(); i++) {
                fileWriter.write(xValues.get(i) + "," + smoothedYValues.get(i));
                fileWriter.write(System.lineSeparator());
            }
            fileWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    // Method to compute moving average
    private ArrayList<Double> movingAverage(ArrayList<Double> yValues, int windowValue) {
        ArrayList<Double> smoothedYValues = new ArrayList<>();
        for (int i = 0; i < yValues.size(); i++) {
            double sum = 0;
            int count = 0;
            for (int j = Math.max(0, i - windowValue / 2); j <= Math.min(yValues.size() - 1, i + windowValue / 2); j++) {
                sum += yValues.get(j);
                count++;
            }
            double average = sum / count;
            smoothedYValues.add(average);
        }
        return smoothedYValues;
    }
}
```

CSVTester

The CSVTester class is designed to test the functionality of the plotting, salting, and smoothing operations.

Below are some of the specifications:

1. Object Creation: Represents objects of the Plotter, Salter, and Smoother classes to perform various operations on the data.

```
// Create objects of Plotter, Salter, and Smoother classes
Plotter plotter = new Plotter();
Salter salter = new Salter();
Smoother smoother = new Smoother();
```

2. Plotting Data: Calls the plot method of the Plotter class to generate and save a CSV file containing the plotted data points.

```
// Plot the data using the Plotter class
plotter.plot("PlottedValues", -10, 10, 0.1);
```

3. Salting Data: Calls the salt method of the Salter class to add random variations to the y-values of the plotted data.

```
// Salt the data using the Salter class
salter.salt("PlottedValues.csv", 0, 250);
```

4. Smoothing Data: Calls the smooth method of the Smoother class to apply a moving average algorithm to the salted data, resulting in smoothed data.

```
// First run: Read from "SaltedValues.csv" and output to "SmoothedValues1.csv"
smoother.smooth("SaltedValues.csv", "SmoothedValues1.csv", 8);
```

5. Multiple Smoothing Iterations: The class is designed to allow multiple iterations of the smoothing process by calling the smooth method multiple times with different input and output file names.

```
// First run: Read from "SaltedValues.csv" and output to "SmoothedValues1.csv"
smoother.smooth("SaltedValues.csv", "SmoothedValues1.csv", 8);

// Second run: Read from "SmoothedValues1.csv" and output to "SmoothedValues2.csv"
smoother.smooth("SmoothedValues1.csv", "SmoothedValues2.csv", 8);

// Third run: Read from "SmoothedValues2.csv" and output to "SmoothedValues3.csv"
smoother.smooth("SmoothedValues2.csv", "SmoothedValues3.csv", 8);
```

Full CSVTester class code:

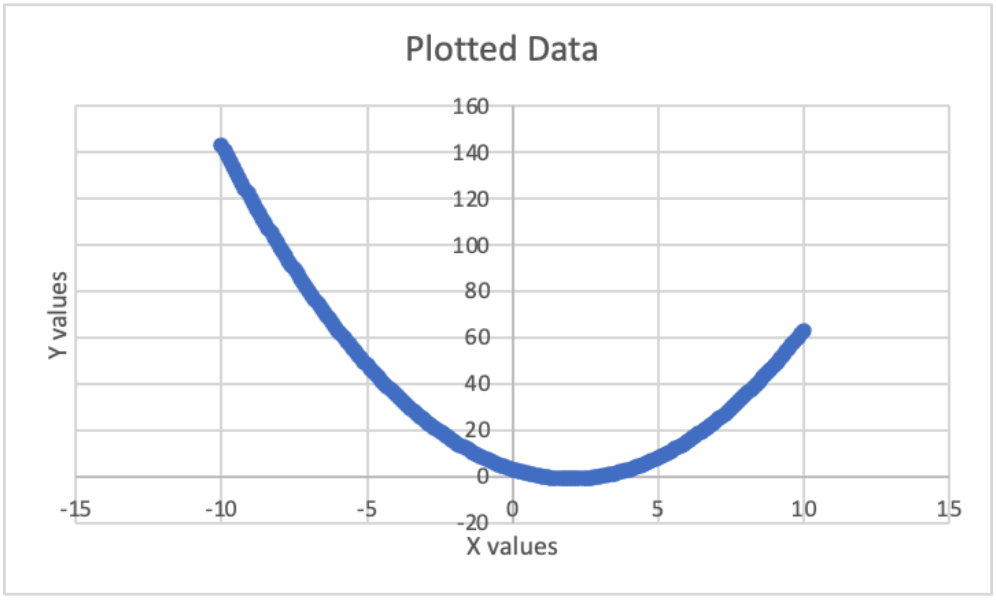
```
public class CSVTester {
    public static void main(String[] args) {
        // Create a new Plotter object
        Plotter plotter = new Plotter();

        // Plot the data using the Plotter class
        plotter.plot("PlottedValues", -10, 10, 0.1);

        // Create a new Salter object
        Salter salter = new Salter();

        // Salt the data using the Salter class
        salter.salt("PlottedValues.csv", 0, 250);

        // Create a new Smoother object
        Smoother smoother = new Smoother();

        // First run: Read from "SaltedValues.csv" and output to "SmoothedValues1.csv"
        smoother.smooth("SaltedValues.csv", "SmoothedValues1.csv", 8);

        // Second run: Read from "SmoothedValues1.csv" and output to "SmoothedValues2.csv"
        smoother.smooth("SmoothedValues1.csv", "SmoothedValues2.csv", 8);

        // Third run: Read from "SmoothedValues2.csv" and output to "SmoothedValues3.csv"
        smoother.smooth("SmoothedValues2.csv", "SmoothedValues3.csv", 8);
    }
}
```

Results - Small Samples

Plotter:

- Data generated using a population range [-10, 10] with an increment of 0.1

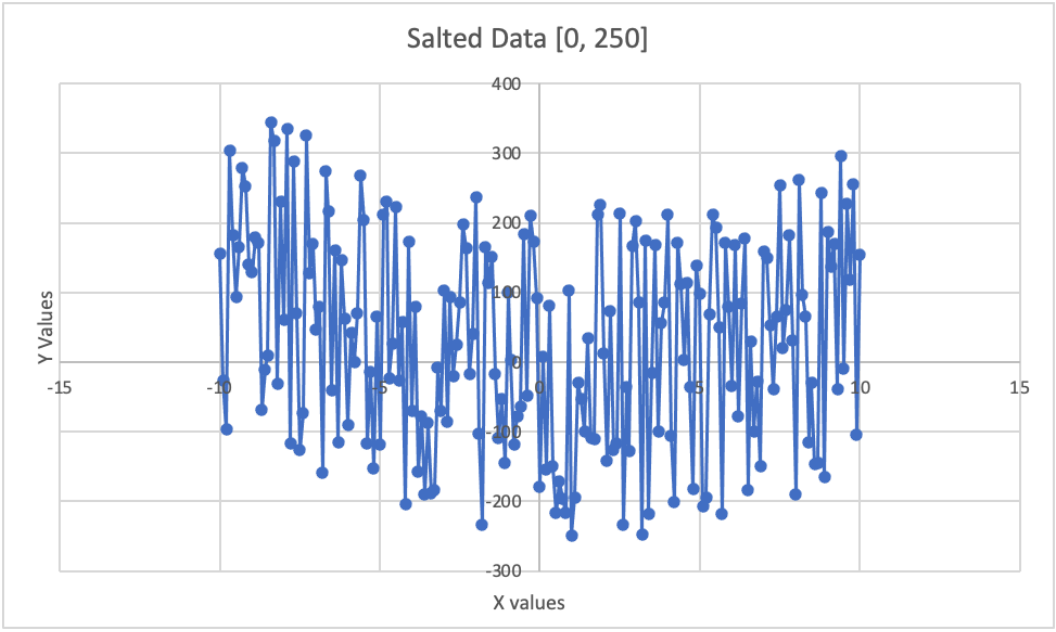
```
// Plot the data using the Plotter class
plotter.plot("PlottedValues", -10, 10, 0.1);
```



Salter:

- Data salted with salt range [0, 250]

```
// Salt the data using the Salter class
salter.salt("PlottedValues.csv", 0, 250);
```

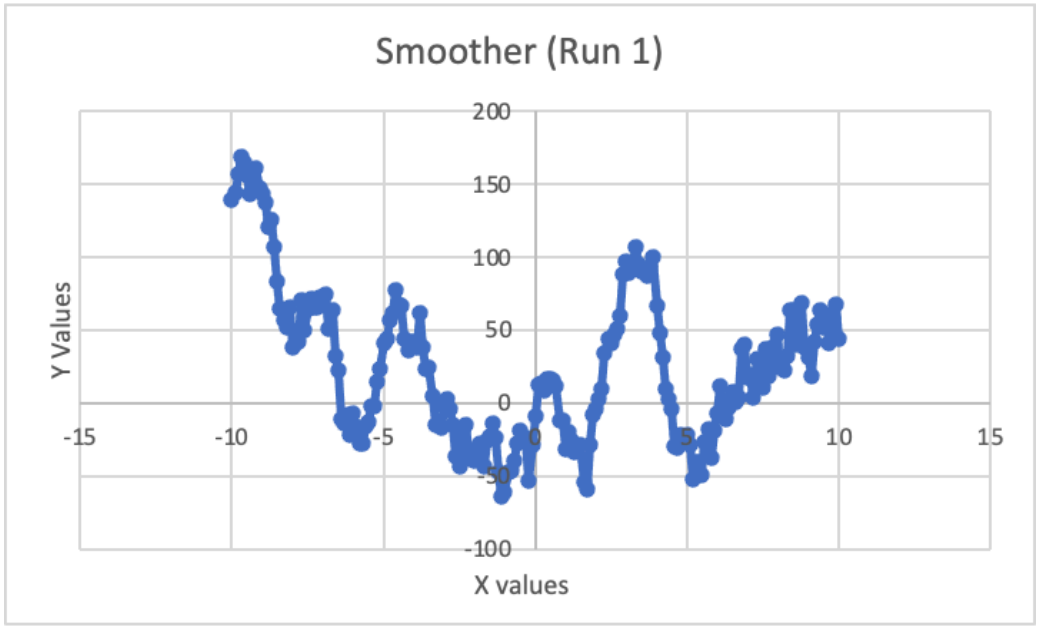


Smoother:

First Run

- Data smoothed with a window size of 8

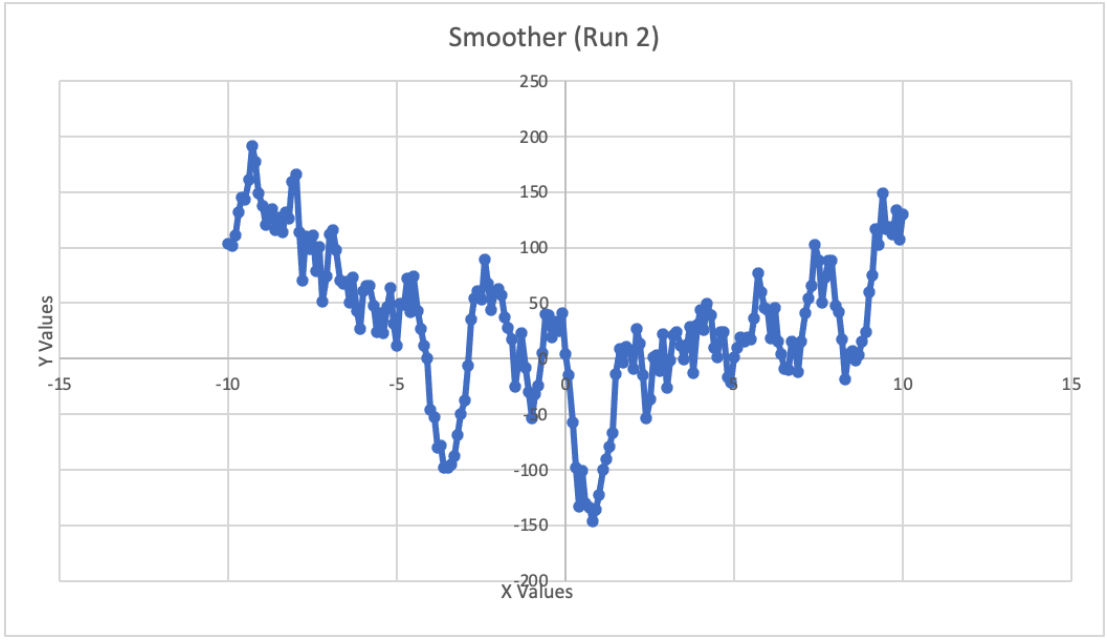
```
// First run: Read from "SaltedValues.csv" and output to "SmoothedValues1.csv"
smoother.smooth("SaltedValues.csv", "SmoothedValues1.csv", 8);
```



Second Run

- Smoother ran on first set of smoothed values

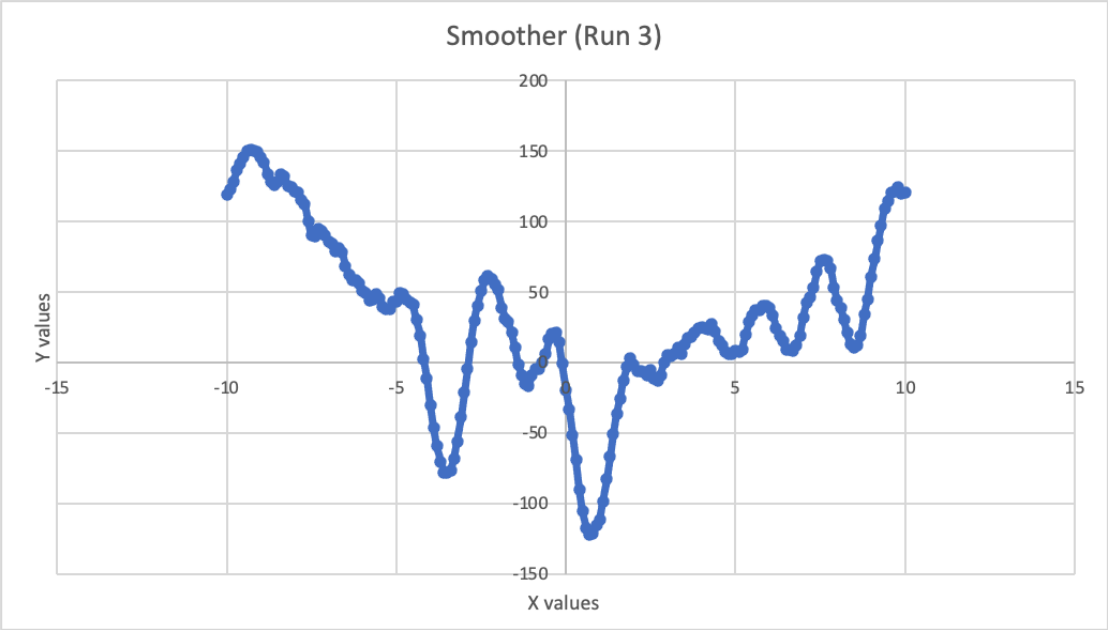
```
// Second run: Read from "SmoothedValues1.csv" and output to "SmoothedValues2.csv"
smoother.smooth("SmoothedValues1.csv", "SmoothedValues2.csv", 8);
```



Third Run

- Smoother ran on the second set of smoothed values

```
// Third run: Read from "SmoothedValues2.csv" and output to "SmoothedValues3.csv"
smoother.smooth("SmoothedValues2.csv", "SmoothedValues3.csv", 8);
```

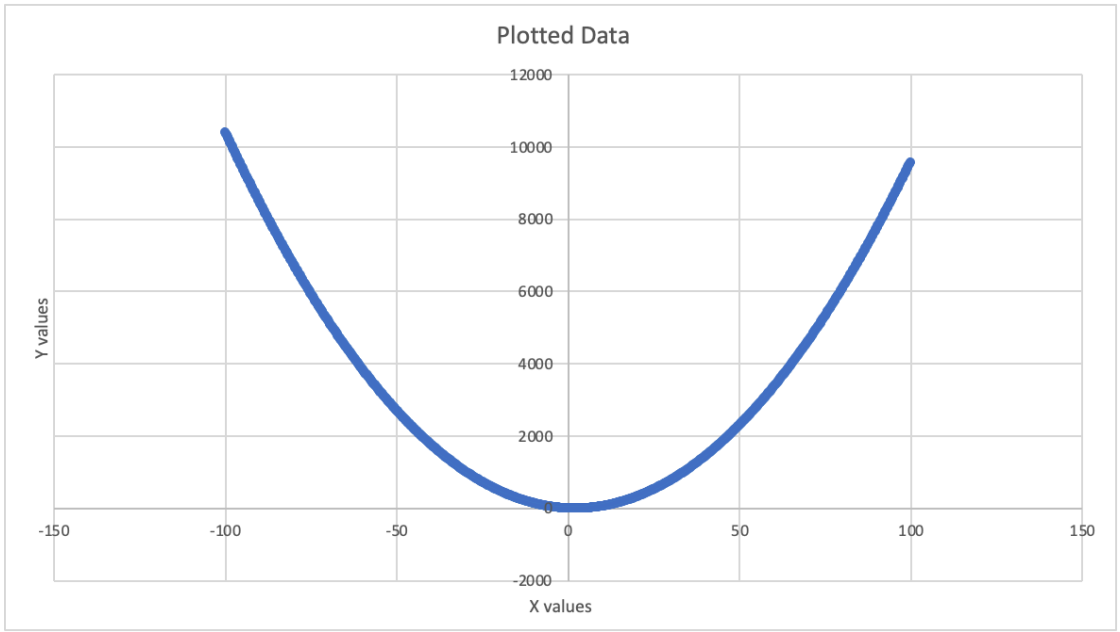


Results - Large Samples

Plotter:

- Data generated using a population range [-100, 100] with an increment of 0.2

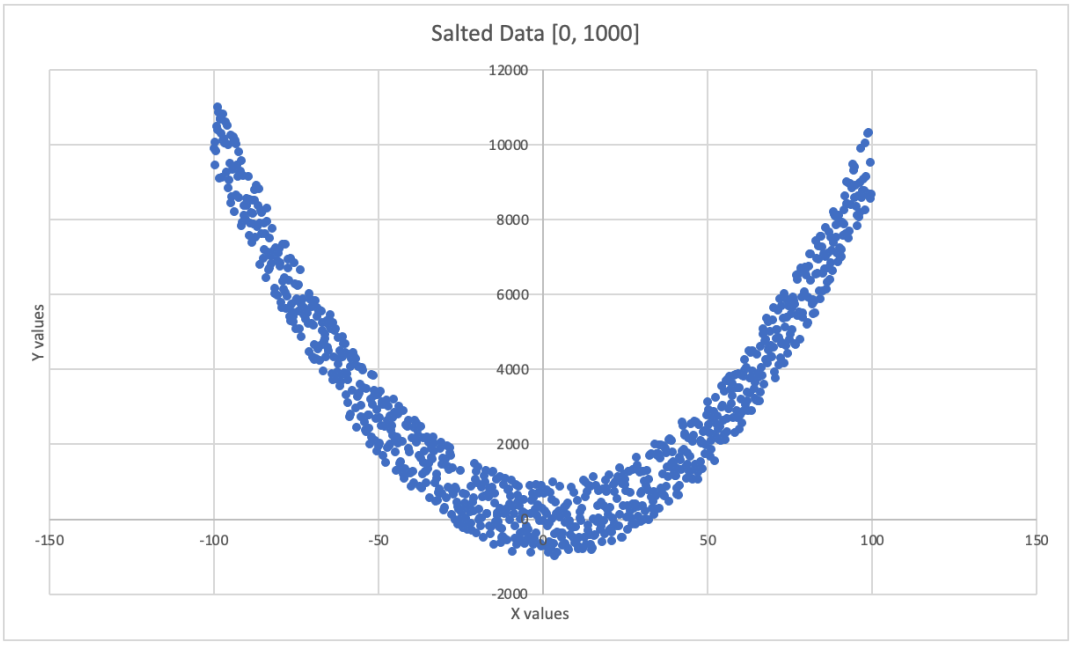
```
// Plot the data using the Plotter class
plotter.plot("PlottedValues", -100, 100, 0.2);
```



Salter:

- Data salted with salt range [0, 1000]

```
// Salt the data using the Salter class
salter.salt("PlottedValues.csv", 0, 1000);
```

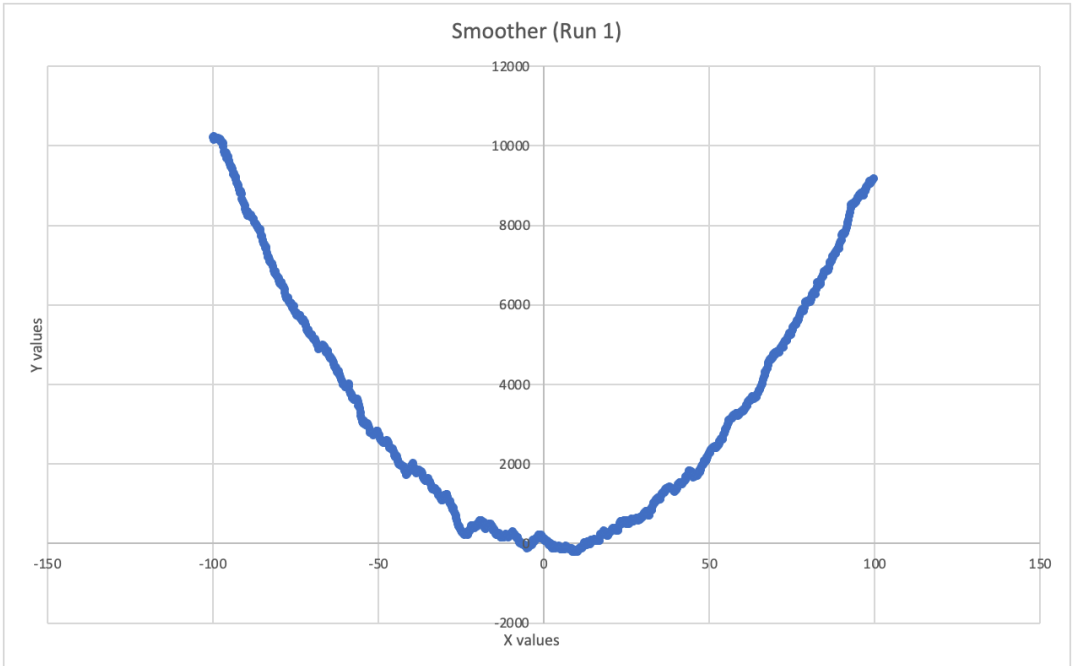


Smoother:

First Run

- Data smoothed with a window size of 20

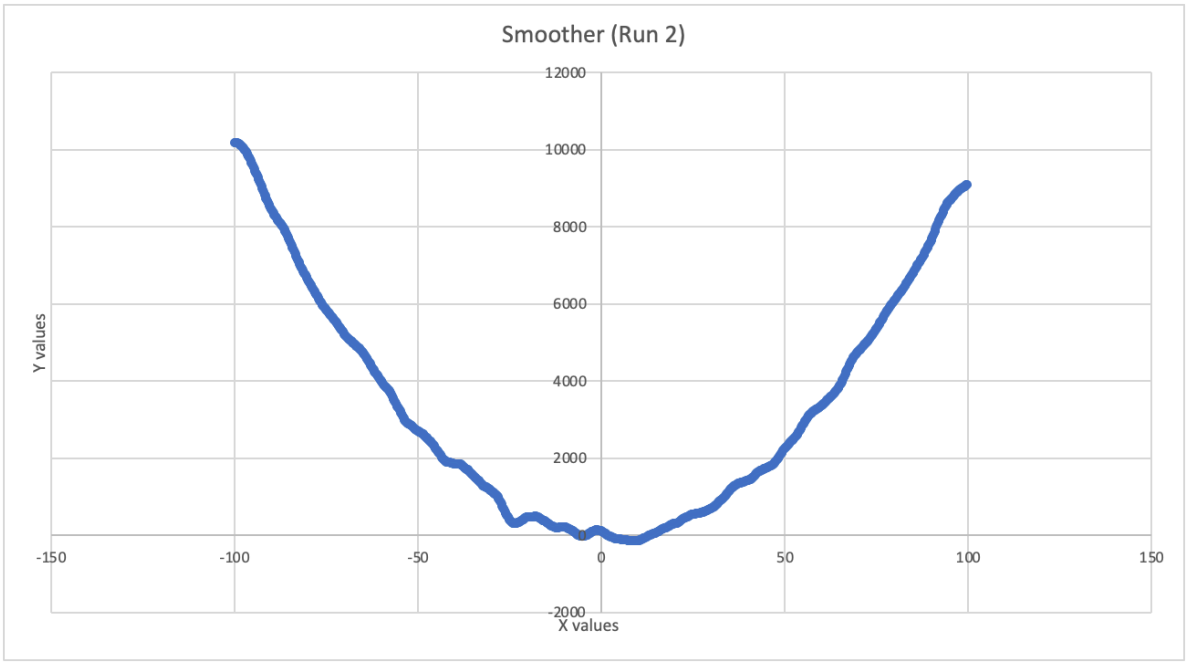
```
// First run: Read from "SaltedValues.csv" and output to "SmoothedValues1.csv"
smoother.smooth("SaltedValues.csv", "SmoothedValues1.csv", 20);
```



Second Run

- Smoother ran on first set of smoothed values

```
// Second run: Read from "SmoothedValues1.csv" and output to "SmoothedValues2.csv"
smoother.smooth("SmoothedValues1.csv", "SmoothedValues2.csv", 20);
```



Third Run

- Smoother ran on the second set of smoothed values

```
// Third run: Read from "SmoothedValues2.csv" and output to "SmoothedValues3.csv"
smoother.smooth("SmoothedValues2.csv", "SmoothedValues3.csv", 20);
```

Discussion

The main objectives of the program were to generate a dataset based on a given function, writing that dataset to a CSV file for storage and analysis, read the data from the CSV file and add some random variation to it (salting), write the salted data to another CSV file, and then read the salted data and apply a smoothing algorithm to reduce the noise introduced during salting, aiming to restore the original shape of the data. To achieve these goals, the program utilized Java programming language to implement classes for plotting, salting, and smoothing data. Each class had specific responsibilities, such as generating data, manipulating it, and saving it to CSV files. Excel was then used to visualize the data, allowing for easy verification of the program's functionality.

The program was tested with small samples of data as well as large samples, for which the quadratic function $y = x^2 - 4x + 3$ was used. The small trial consisted of a population size of 200 data points, ranging from -10 to 10 with an increment of 0.1. The plotter successfully made a table of the function, using the given data points, saving it as a CSV file. This file was then opened in Excel and graphed to visualize an upward curving slope. After salting the data with a salt range of 0 to 250, the graph now looked different, with data points all over the place. There was a zig-zag pattern in the curve. After the first smoother run, the graph looked cleaner, as it calculated the smoothed y-values using a moving average algorithm. Although the graph trendline was still a bit bumpy, it got smoother with each run of the program.

Similarly, the second sample was tested the same way, but with more input values. This trial consisted of a population of 1000 data points, ranging from -100 to 100 with an increment of 0.2. Once plotted, the graph for this set of data looked more complete. The curve was now shaped almost like a smile. The data was salted with a salt range of 0 to 1000, which resulted in a lot more variability in the y-values. Like the first test, the smoother was run a total of three times, with each one smoothing the previously produced CSV file. As expected, the data became more consistent and clear with each run. After the third run, the graph almost looked the same as it did when it was first plotted. The program seemed to achieve better results with more data points.

There’s many ways the programs could be modified for efficiency. Firstly, more sophisticated salting techniques beyond simple random variation could be used. Also, the salter class could allow users to decide how much of their data gets changed, rather than changing every other point. In the plotter class, the program could ask users to input custom equations when it is run rather than being limited to a fixed function. In the smoother class, the smoothing algorithm could be enhanced to support different types of filters and not just the moving average. The program could generally be improved to record important events and errors during data processing for better troubleshooting and analysis with logging functionality.

In conclusion, apart from accomplishing the primary goals of plotting, salting, and smoothing, the program also emphasized the significance of data processing in many fields such as scientific research, financial analysis, and statistics. By offering an organized method for processing data, the application showed how repetitive activities can be automated and streamlined. Overall, the project served as a practical example of how programming languages

such as Java may be applied to address realistic data processing issues and provided insightful knowledge of the concepts and procedures of probability and statistics-related data manipulation and analysis.