Christopher Ward

Probabaility and Statistics

JFreeChart/Apache Math Library Plot Salt Smooth Documentation

## Introduction/Installation

The first thing I did was google JFreeChart and go to their website ([here](https://www.jfree.org/jfreechart/)). However, I was only able to download a .zip or .tar file from the official website and I wanted to find a .jar to easily include in my referenced libraries for VS Code, which is my personal IDE of choice. It took me a little while, but eventually I learned that many versions of the Maven repository include JFreeChart. So, I went to the Maven central repository files ([here](https://central.sonatype.com/artifact/org.jfree/jfreechart/versions)) for JFreeChart and found the most current version of JFreeChart I’ve seen and in a .jar format, perfect for my use.

Once I got the JFreeChart .jar installed I moved on to getting a .jar of the Apache math commons library. Again, I went to the Maven central repository ([here](https://central.sonatype.com/artifact/org.apache.commons/commons-math3/versions)) to find the library I needed in a .jar format. I could always have built a new Java project with Maven tools, which would’ve included both libraries, but I had already set up project files within my project2 folder, and I thought this would be easier than restarting just to build with Maven.

Finally, after both .jar were installed and under my referenced libraries in the java project I was ready to begin plotting.

## Plotting the Function

The first step in the Plot, Salt, Smooth (PSS) process was to generate and plot my original function. I began by defining the function as over a specified range of x-values. This was achieved in the ‘Plot’ class, where I utilized JFreeChart’s ‘XYSeriesCollection’ and ‘ChartFactory’ to create an XY line chart. The ‘XYSeriesCollection’ is great because over time I can add more function to the dataset (i.e. salted/smoothed function) and plot the entire collection together on one chart. I initialized the range from -10 to 10 with a step size of 0.1 to make sure a lot of values are included for salting. I used Java's ‘Function’ interface. I also use a JFrame to display my newly created chart.

A black background with white text

Description automatically generated

I was satisfied with how the first graph came out in JFreeChart. However, I wanted to make my program not have a hardcoded function to plot. In the first part of the plot, salt, smooth assignment I used the ‘exp4j’ library, which is ‘a simple mathematical expression evaluator for java” ([MVN Repository](https://mvnrepository.com/artifact/net.objecthunter/exp4j/0.4.8)). With exp4j I can represent my function expression as a string and have the evaluator do the function calculations for me. I also wanted to limit Plot.java to only plotting, so I abstracted out my expression builder and evaluation to a ‘Calculations.java’ class.

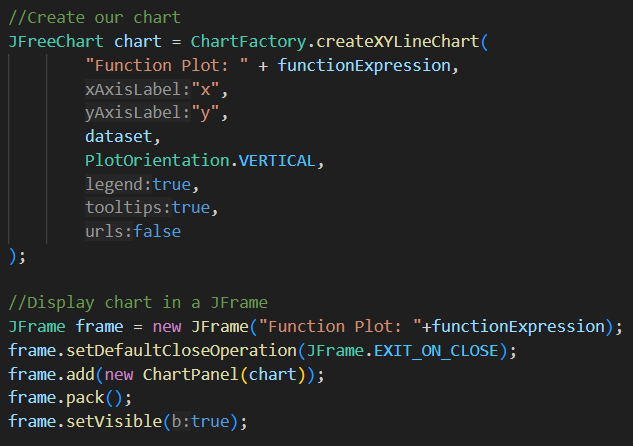
With my expression builder working properly and in it’s own class for calculations, I decided to try a slightly harder function to test my program, so I did: . The actual plotting was done using the ‘Plot’ class, which utilizes JFreeChart to display the graph. The ‘displayGraph’ method of the ‘Plot’ class took the x and y values and rendered the function as a simple line chart.

I’ve included screenshots of my calculations class and the original function I plotted,

, the second function , and snippets of my code for reference:

A computer screen shot of a program code

Description automatically generated



\*\*First function I plotted\*\*

A graph with a red line

Description automatically generated

\*\*Second function I plotted\*\*

A graph with a red line

Description automatically generated

## Salting the Function

Once the original function was plotted, I moved on to salting the data. This step was fairly simple as Apache Commons Math introduces gaussian random generation to simulate real-world imperfections and variability. The ‘Salt’ class handled this process by applying Gaussian noise to the y-values. I used the Apache Commons Math ‘GaussianRandomGenerator’ to generate the salt values. The strength was configurable, and I set it to a moderate value (5) for clear visualization. In the ‘salt’ class, the ‘addSalt’ method iterated through the y-values, adding salt value to an array list. I then added the list of salted values to our XYSeriesCollection and ran my plotter again:

A graph showing a function

Description automatically generated

The red is my original function, and the blue jagged line overtop is my original function with the salt values add to the y-values.

My entire Salt class is below:

A computer code on a black background

Description automatically generated

## Smoothing the Function

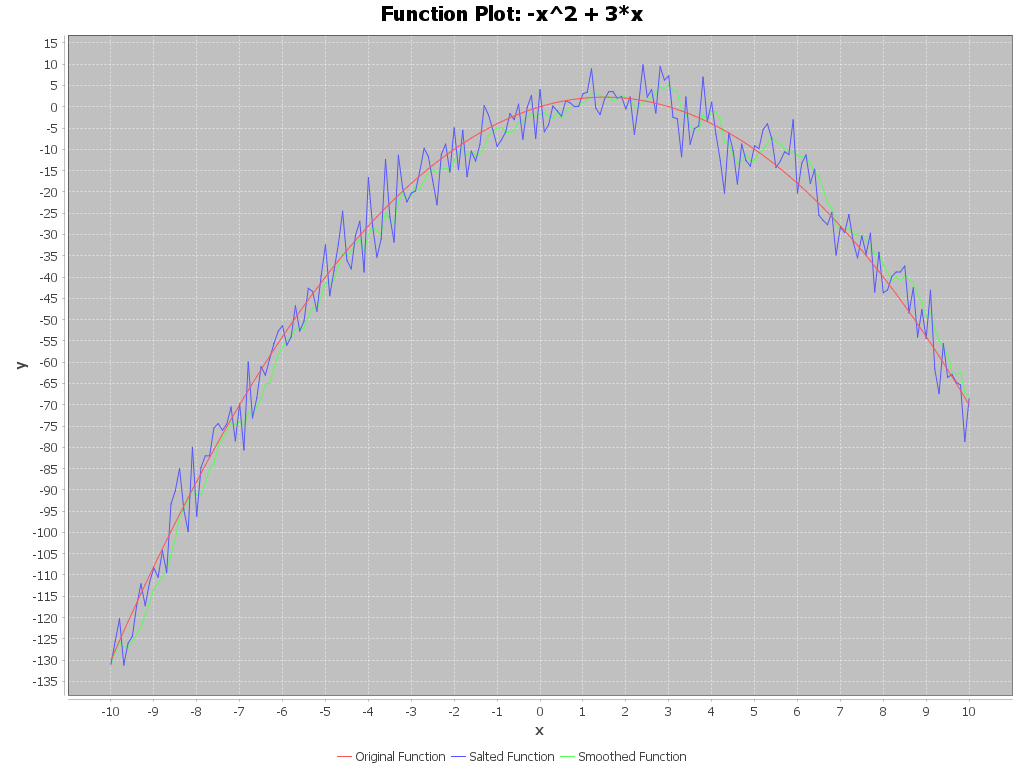
The final step in the PSS process was smoothing the salted data to approximate the original function. I implemented this step in the ‘Smooth’ class using a moving mean algorithm. The smoothing operation reduced the noise while preserving the overall trend of the data.

The ‘applyMovingMean’ method in the ‘Smooth’ class performed the smoothing. Using the Apache Commons Math ‘Descriptive Statistics’ class, I calculated a moving average over a specified window size, set to 5 in this case. I used the documentation on Descriptive Statistics on the commons apache website ([here](https://commons.apache.org/proper/commons-math/commons-math-docs/apidocs/org/apache/commons/math4/legacy/stat/descriptive/DescriptiveStatistics.html)) to help set up a smoother with setWindowSize() and getMean(). With this approach I can calculate the mean for any given window of the function to get a ‘moving mean’ and use this to smooth my function.

A screen shot of a computer code

Description automatically generated

In the ‘smooth’ class I applied the moving mean based on our current window to the salted y-values. Again I added the new smoothed values to an array list, which I used to make our series for the XYSeriesCollection. Overlaying the smoothed y-values yielded a result of smoothed values that didn’t resemble the original function that well.



The green line shows my smoothed values, but there is a lot of variation between the original and the smoothed function. I believe the reason for this lies within the values for the strength of my gaussian random number generator and my smoothing window. Originally, I had multiplied my salt value by 5, to make it easier to see the changes in my graph. However now problem was my smoother wasn’t strong enough to counteract the strong salt values. After making the change to my salter, the graph came out as one I was much more satisfied with. The final smoothed graph is slightly shifted in the positive y direction, but other than that it does closely resemble the original function we plotted. At least, it is a much closer to our original function than the first smoothed function was.

A graph of a function

Description automatically generated

## Closing Remarks

This project was a fun way get more practice with plotting, salting, and smoothing data using Java. By working with JFreeChart, Apache Commons Math, and the xp4j library I created a program that could plot functions, salt the data and then try to smooth it back out to get the original function.

Through this project, I gained a deeper understanding of how to integrate external libraries, such as .jar files into Java projects and utilize tools like Gaussian random number generation and moving averages. While initial challenges such as managing salt strength and smoothing accuracy arose, I believe I managed to achieve a result that closely resembled the original function. This process highlights the importance of fine-tuning parameters to balance realism and precision in data manipulation.

Overall, the PSS project underscored the practical value of combining mathematical principles with programming to visualize and analyze data in innovative ways. This journey not only improved my technical skills but also reinforced my ability to troubleshoot, adapt, and refine solutions—skills essential for tackling complex computational problems in the real world.