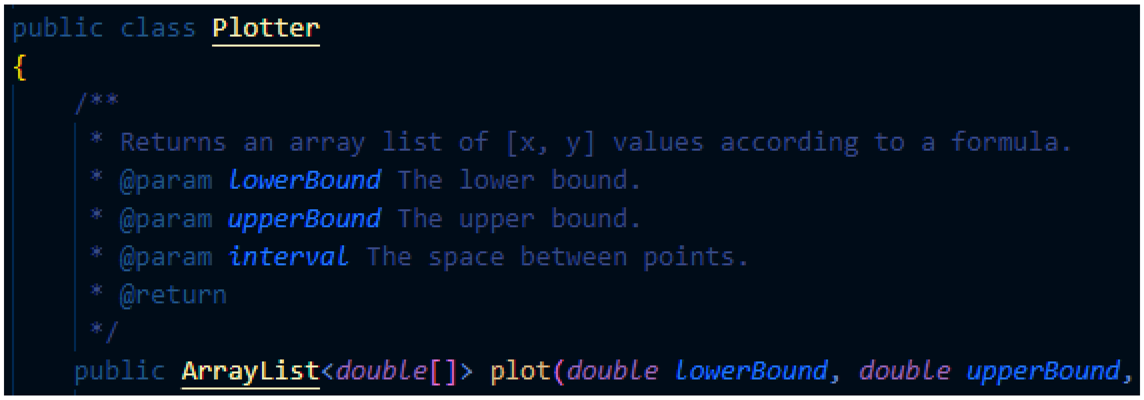
PSS 1 - Introduction

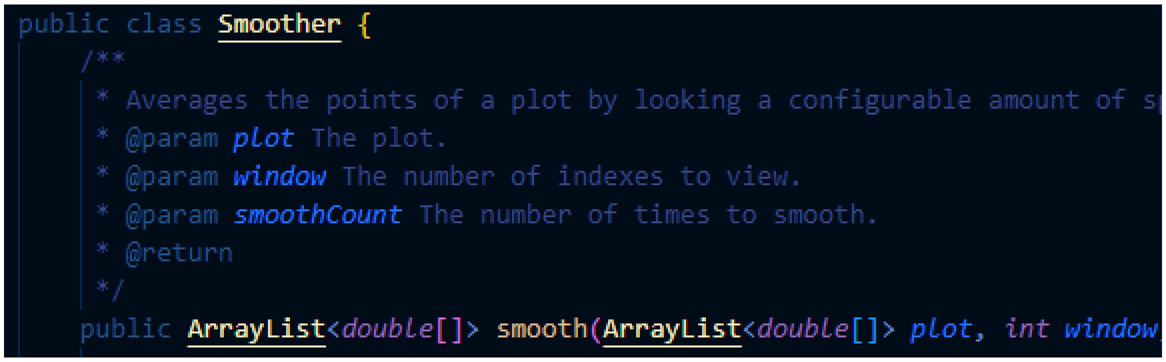
For the PSS part of the project, I chose a rather simple function: x^3. I've also compiled each of the three reports into this singular document. Before we do any graph analysis, we can shortly go over the first part of the PSS assignment, the code. I created 5 classes to perform the function of the assignment:

- Plotter: This takes in a lower and upper bound as well as the interval between points. It returns an array of all points in those points that plot the function x^3.
- Salter: This takes in an array of points from above, the plot, as well as a "salt range" to randomize the y-values of the plot.
- Smoother: This class takes in a plot, intended to be salted already, and also a "window" value. It goes through each x-value, looks "window" spaces to the left and right, and sets the y-value to the average of those windows. There is also a loop count for smoothing multiple times.
- Writer: This takes in a plot and a file name and writes it to an .csv file.
- Finally, SaltSmoothRun creates several .csv files of the graph at different stages of the entire process, with different parameters set at the start.

Every class and method has proper java documentation, as these snippets show:



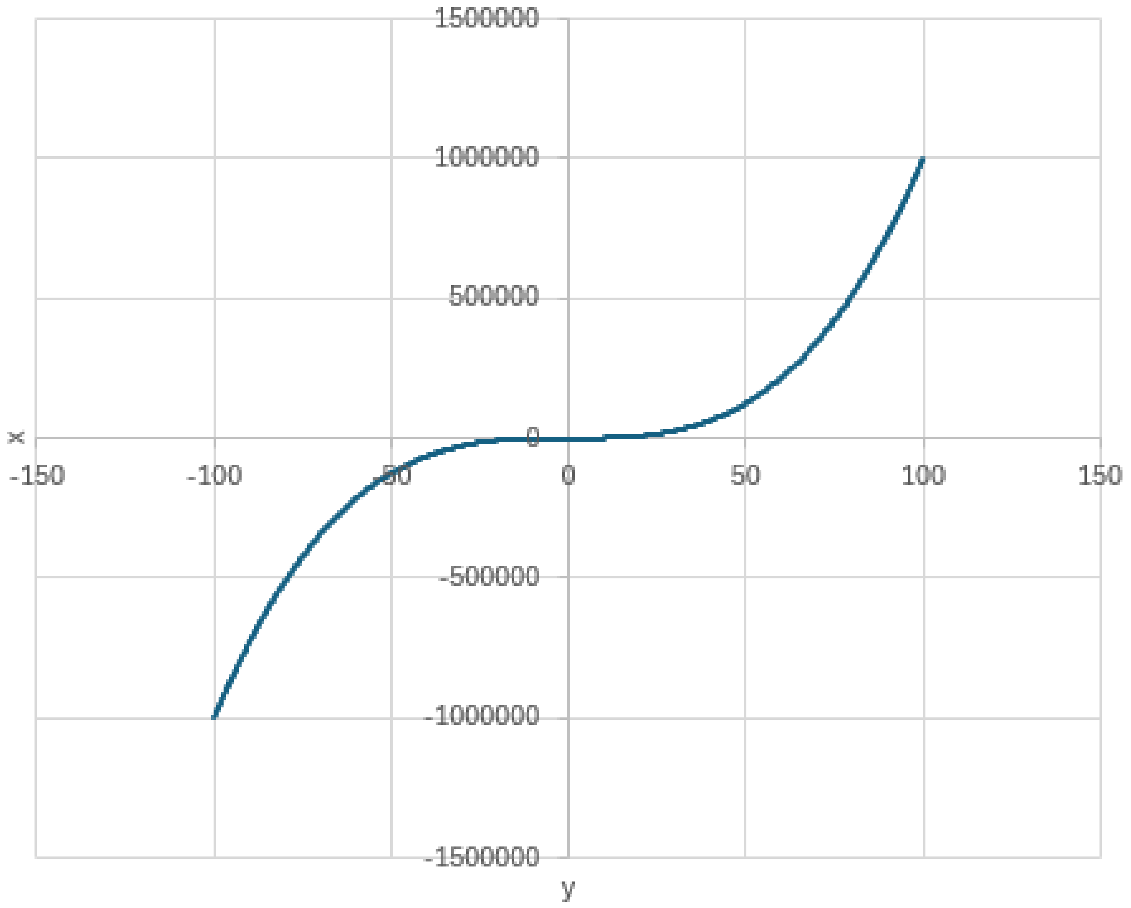
```
public class Plotter
{
    /**
     * Returns an array list of [x, y] values according to a formula.
     * @param lowerBound The lower bound.
     * @param upperBound The upper bound.
     * @param interval The space between points.
     * @return
     */
    public ArrayList<double[]> plot(double lowerBound, double upperBound,
```



```
public class Smoother {
    /**
     * Averages the points of a plot by looking a configurable amount of s
     * @param plot The plot.
     * @param window The number of indexes to view.
     * @param smoothCount The number of times to smooth.
     * @return
     */
    public ArrayList<double[]> smooth(ArrayList<double[]> plot, int window
```

And now, we can look at various graphs created by these classes and analyze how different parameters can effect how the results appear.

Plotting

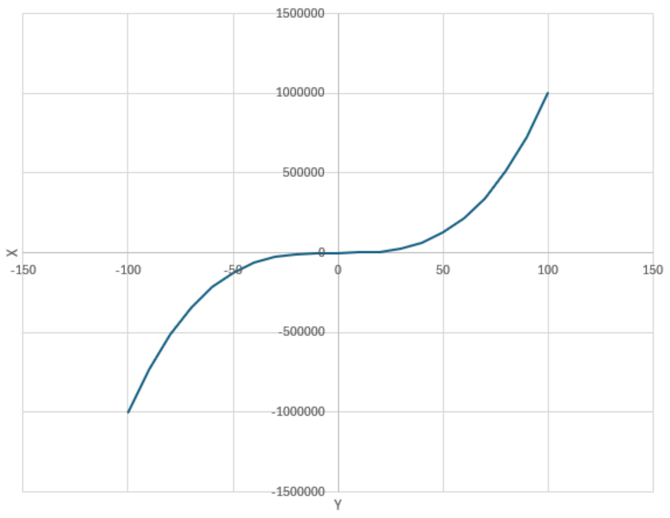


[-100, 100]
Interval: 0.1

Graphs were attained as instructed: .csv files were to be opened within Microsoft Excel and a chart was made where points were connected via a straight line. I've made an effort to ensure that the chart resembles a coordinate plane, as Excel does not offer that within its default options.

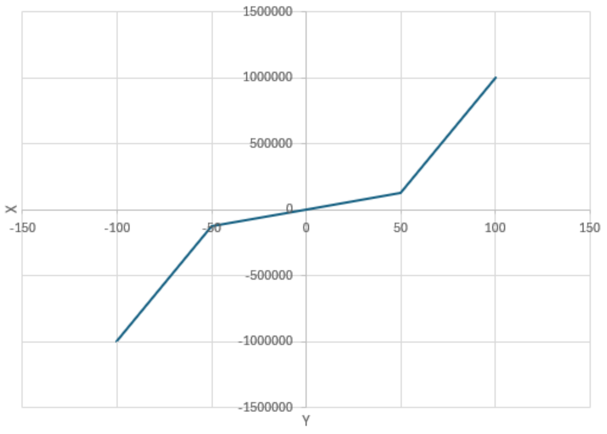
Pictured is a plot of our function, completely unmodified. All relevant parameters have been placed below each graph to easily connect the two. We can see here that our plotter is working well and smoothly. It's exactly what one would expect from a plot of x^3.

We will now see how changing the parameters affect it's appearance.



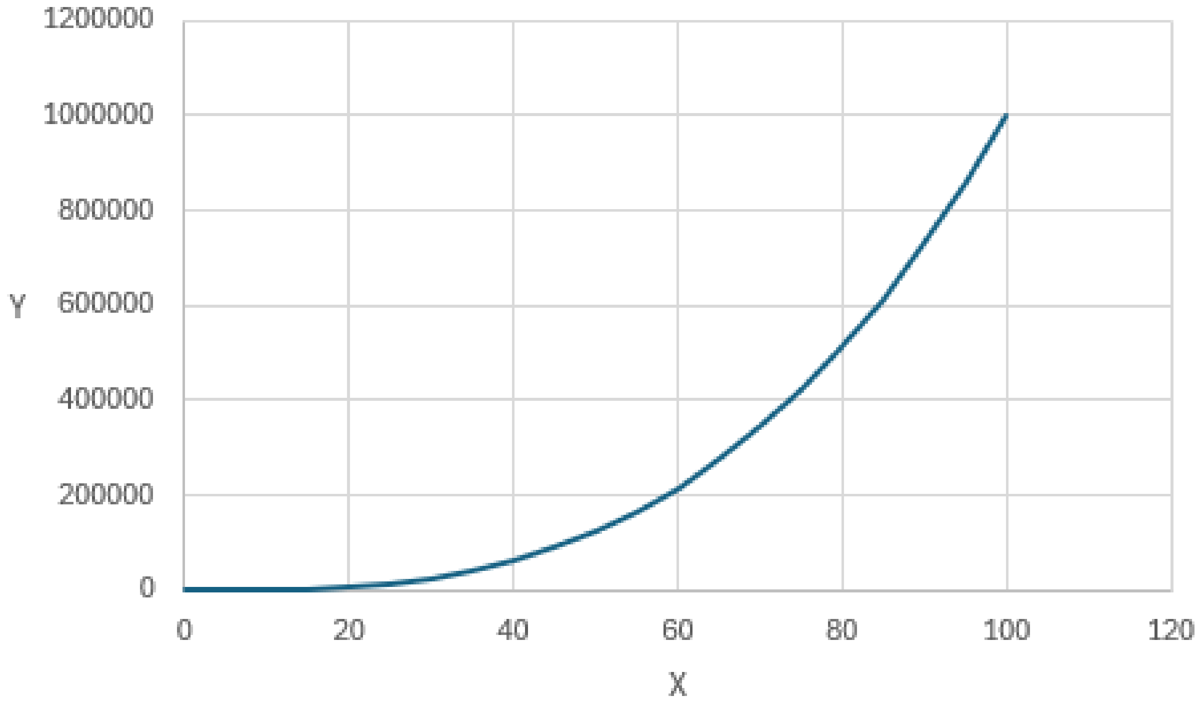
[-100, 100]
Interval: 10

Though it may be hard to notice, if one were to closely examine this graph and the previous one side by side, the difference would be clearer. This one uses an interval of 10 rather than the previous 0.1, giving a much smaller "resolution" of the graph and, as a consequence, holds about 100 times less data points than before. You may notice that the areas with a high rate of change look jagged, as if they were drawn with a low polygon count.



[-100, 100]
Interval: 50

We can push this effect to its limit by enforcing an interval of 50. Because of the bounds, this graph now only contains 4 points, and by extension, 4 lines. It's an extremely rough approximation of the function, but it goes without saying that the functionality of the parameter works exactly as expected.

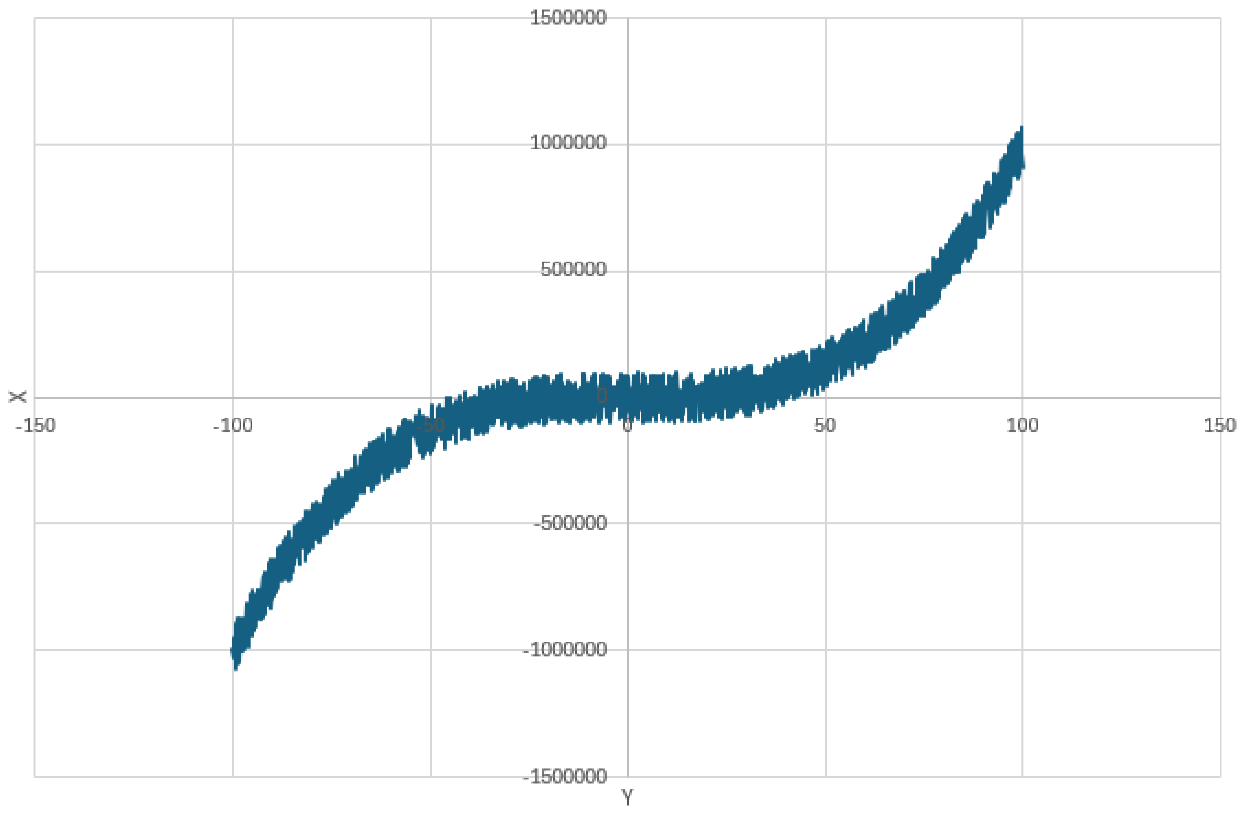


[0, 100]
Interval: 5

We move on from changing the interval and test the functionality of what happens when the bounds are changed, now starting from 0 rather than -100. As expected, the graph no longer tracks any negative data and only half of what was left remains.

There’s not much more testing to be done in this aspect. The plotter, evidently, serves its entire purpose with ease. The real meat-on-the-bone comes in once we look into the salter and smoother.

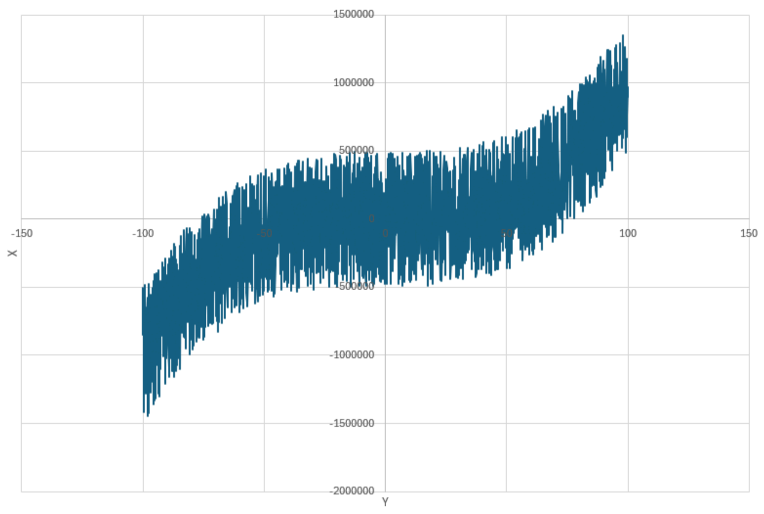
<u>Salting</u>



[-100, 100], Interval: 0.1
Salting: +-100000

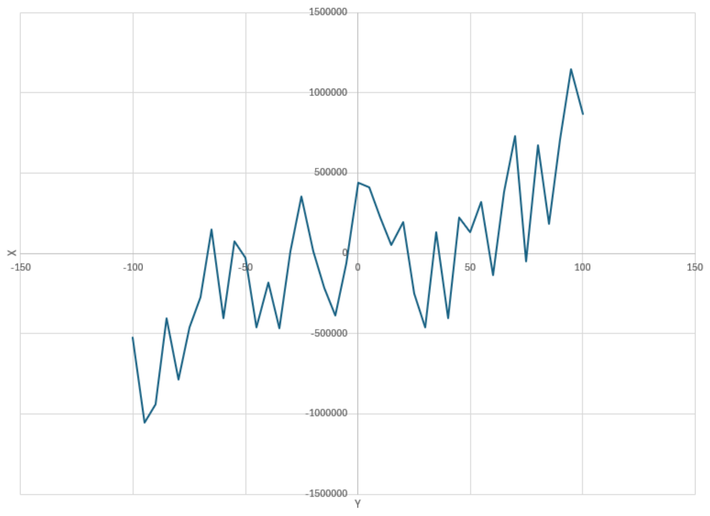
Pictured is our first graph of salted data. With bounds from -100 to 100 and an interval of 0.1, there should be about 2000 datapoints here. Since each one of those has been randomized by adding +-100000 to it, the values between points changes drastically and quite rapidly, causing the plot to look "thick". However, this is merely an artifact of Excel's graphing process. Removing the line, one would see something akin to a scatter plot.

Due to the comparatively small salting range compared to the range of the y-values, the form of the original function is still obvious.



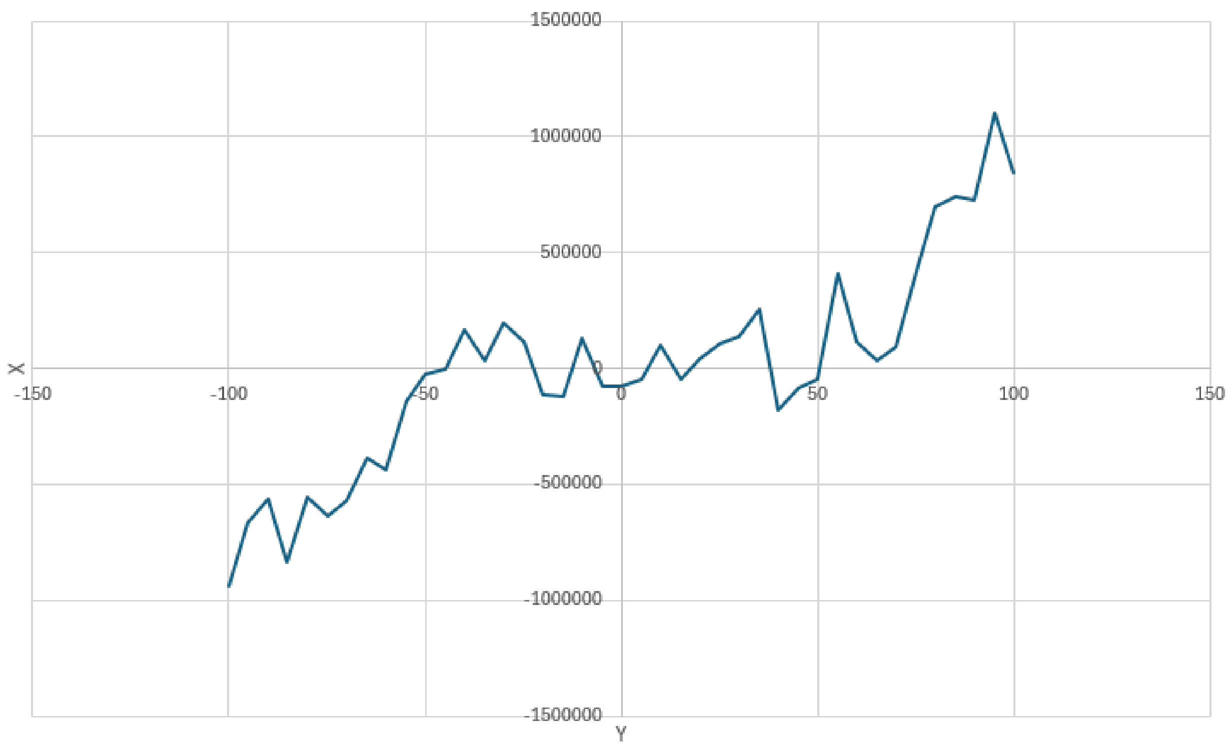
[-100, 100], Interval: 0.1
Salting: +-500000

Raising the salting range a considerable amount, the plot becomes much more chaotic and the form of the function slowly becomes more and more fuzzy.



[-100, 100], Interval: 5
Salting: +-500000

Lowering the interval and thus the amount of data points there are, we can much more clearly see how chaotic the variation between points really is. With only this much information, one could expect smoothing to only work so well. An approximation to the original form becomes much more difficult with less points to work with, as the number of data points helps mitigate the randomness.

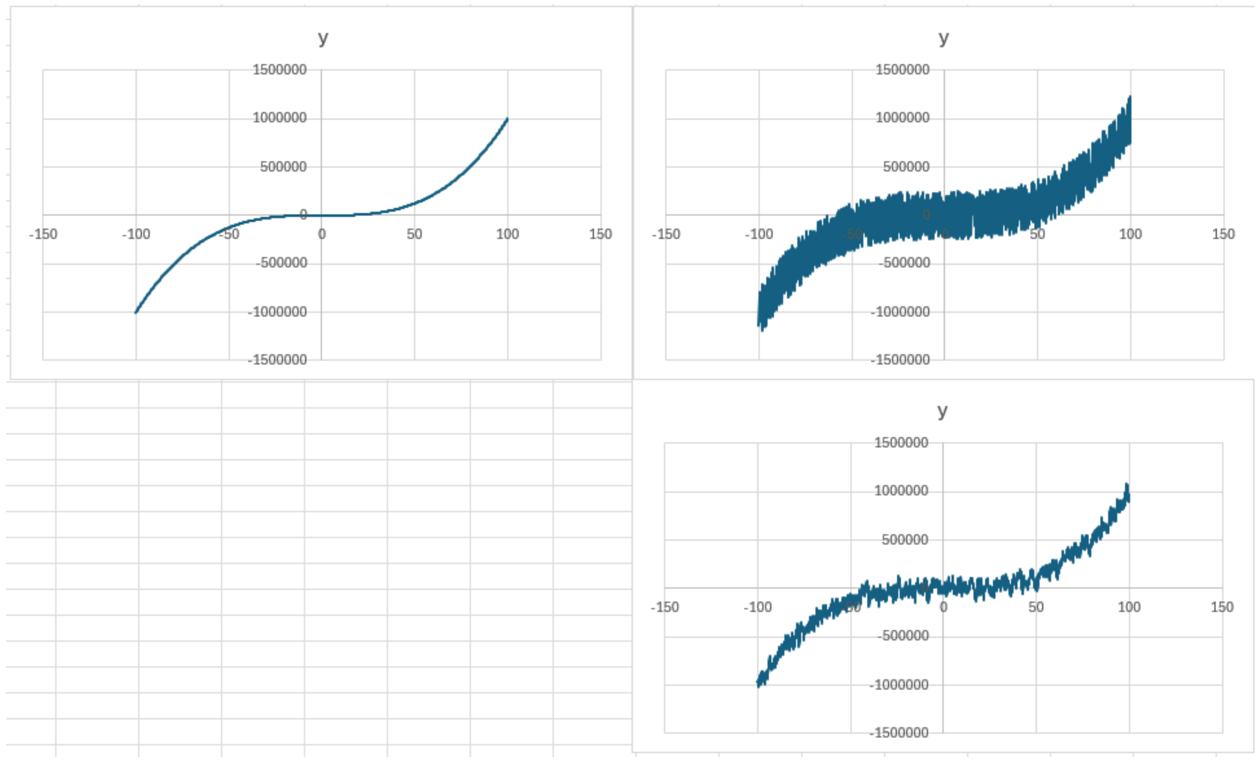


[-100, 100], Interval: 5
Salting: +-250000

We've lowered the salting range by half, and can see how it still resembles the original function.

The salter works as expected, but we've noticed how salt range and the number of points could possibly affect the efficacy of smoothing the data. With a tiny interval, the amount of data points can be averaged better and can mitigate the randomness. However, with a larger interval, mitigating the randomness becomes much more difficult and the original form could be lost.

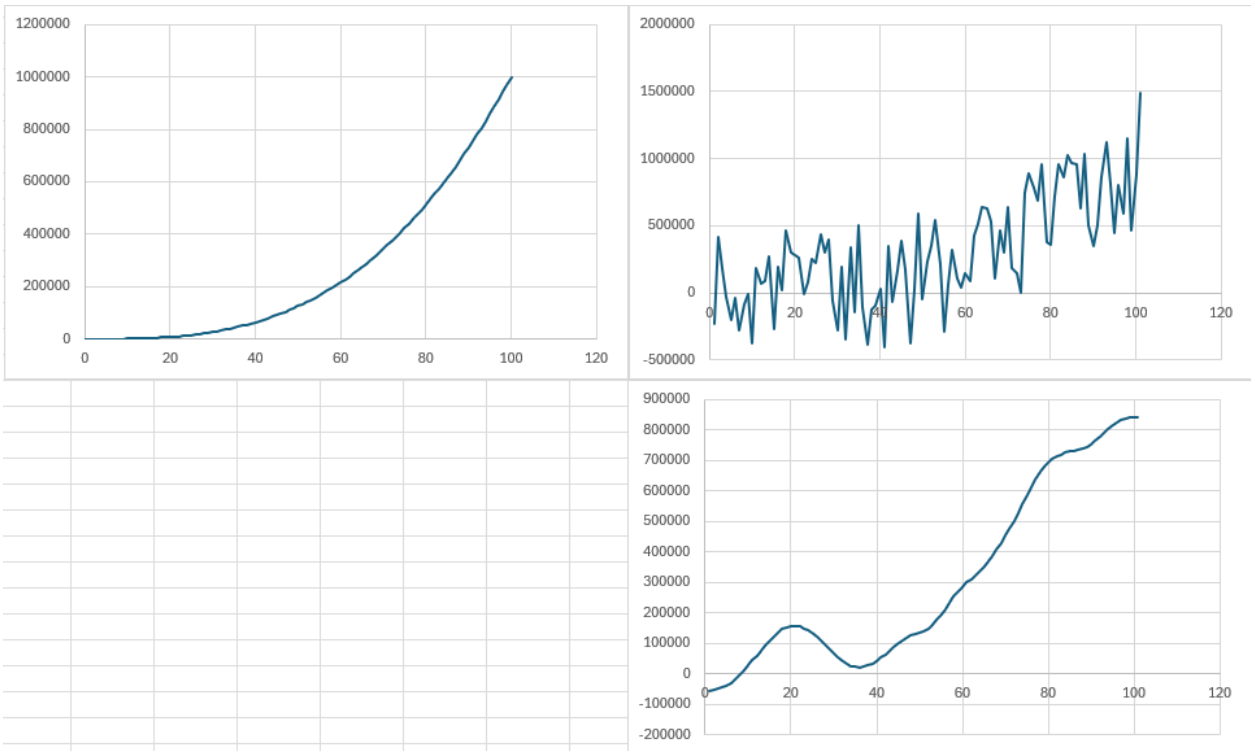
Smoothing



[-100, 100]
Interval: 0.1
Salt Range: 250000
Window: 3
Times smoothed: 1

I've compiled all three stages of PSS into here. In the top left is the original plot. In the top right is the salted data, and the bottom right is the data after being smoothed.

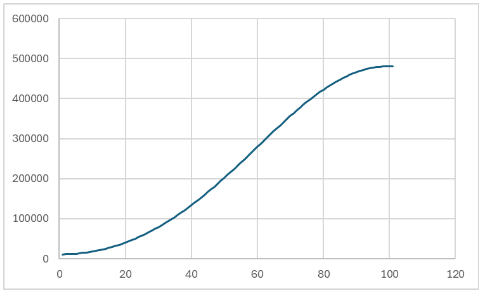
We used a window value of 3 and only smoothed it once. However, we can still see that even after smoothing, the data is still remarkably jagged. It's approached the original function, but it still has a large degree of chaos.



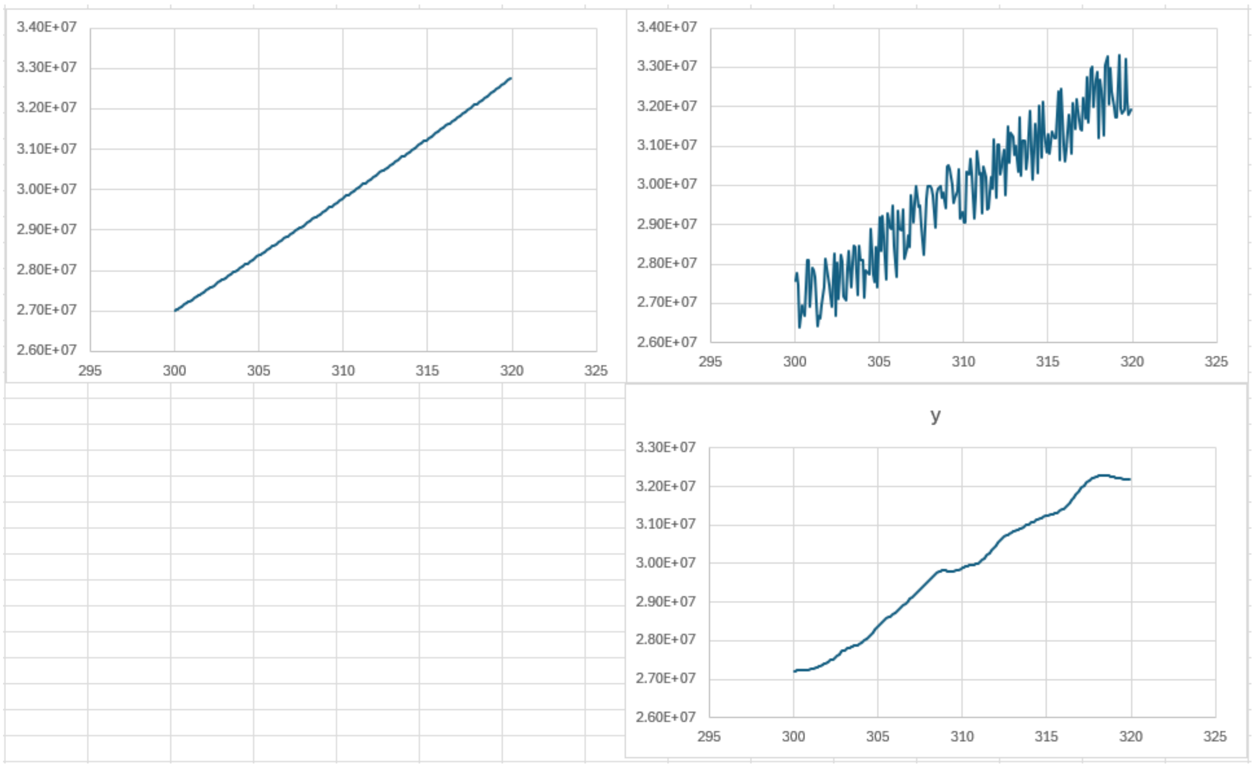
[0, 100]
Interval: 1
Salt Range: 500000
Window: 3
Times smoothed: 3

We've narrowed the bounds and the interval to get a closer look at how data with a comparatively low amount of data points could possibly not accurately represent the original data when smoothed. Considering how previously, smoothing once still showed jaggedness, I've also increased the smoothing count to 3.

We can see that the smoothed function is rather wavy, has bumps, and even has different concavity when compared to the original function. It looks more like a wave rather than a cubic curve. Because of the low data points, we can see how areas where salting has rose a general area higher causes it to smooth out as a heap, seen at x=20. We can try to circumvent this by smoothing it more and more, but a smooth count of 100 shows this:

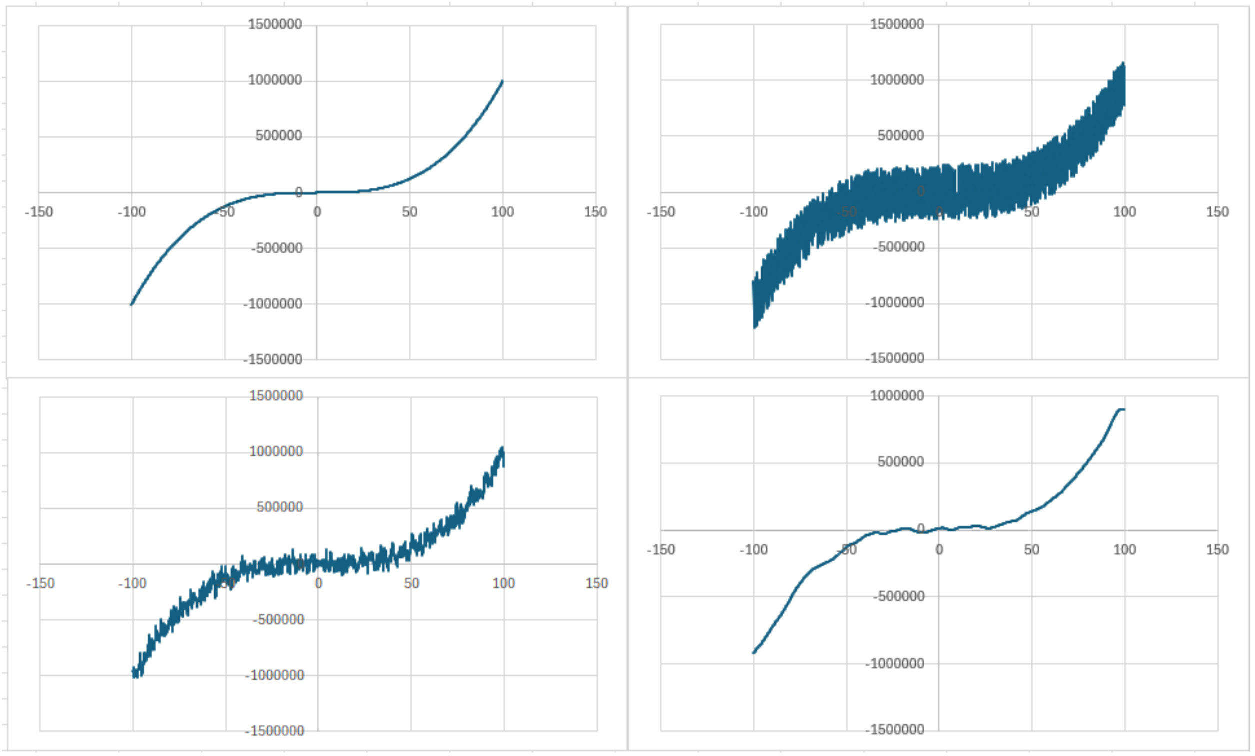


Likely due to the technique of smoothing, the graph now much more resembles a wave rather than a cubic curve. We can see how excessive smoothing will likely not return an accurate result as it will also start smoothing out the original form.



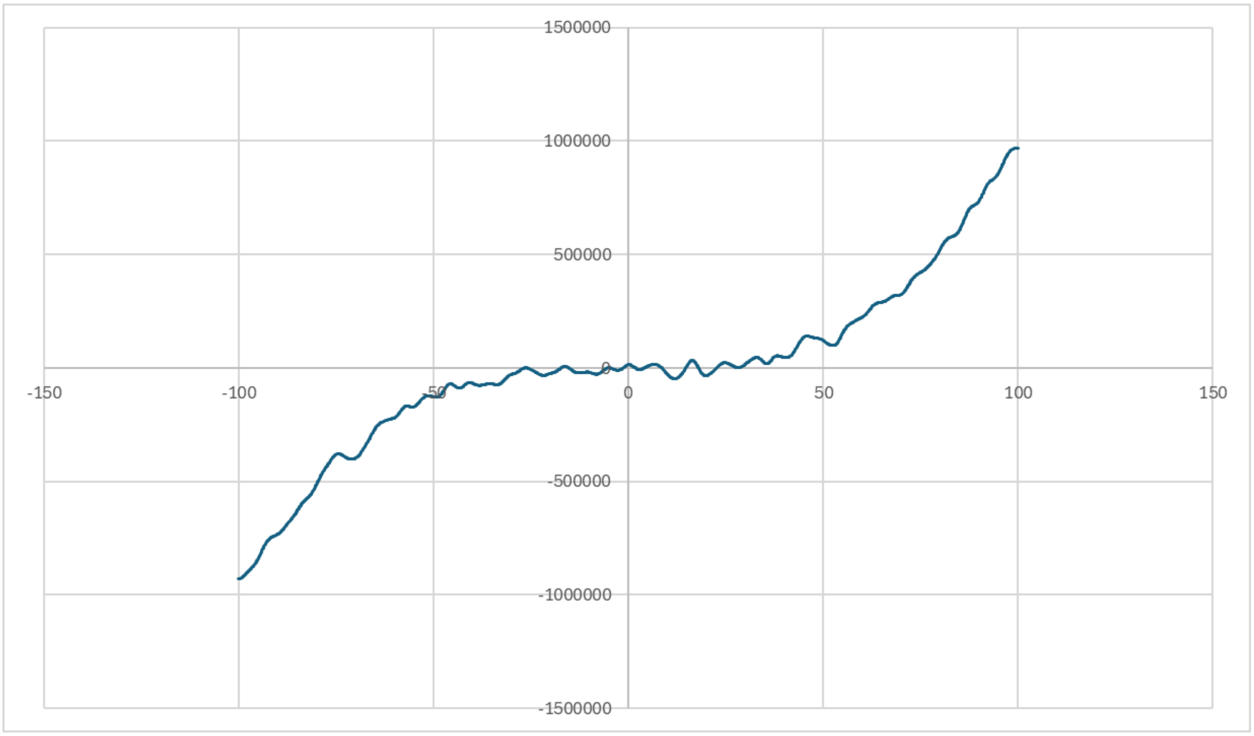
[300, 320]
Interval: 0.1
Salt Range: 1000000
Window: 3
Times Smoothed: 5

Here, I pick a new set of bounds that roughly show the plot as a straight line in order to see how well the smoothing would get to it. You can see how it does a rather good job, (very) roughly approximate to a straight line. However, you can see the consequences of excessive smoothing start to appear, as te minimum and maximum values on the left and right have moved from their original location, and also have started to curve as though it were a wave function. I believe this curving occurs because when smoothing is performed on the edges of the graph and one of the sides of the window is out of bounds, it skews the data to the other side.



[-100, 100]
Interval: 0.1
Salt Range: 250000
Window: (Bottom Left: 1, Bottom Right: 30)
Smoothing Count: 3

The top left and the top right are still original and salted respectively, but the bottom corners now represent different smoothed data with a different window value. They've each only been smoothed 3 times, but you can see that the data for a window of 1 is very jagged whereas the window of 30 is much more smooth with a far less work.

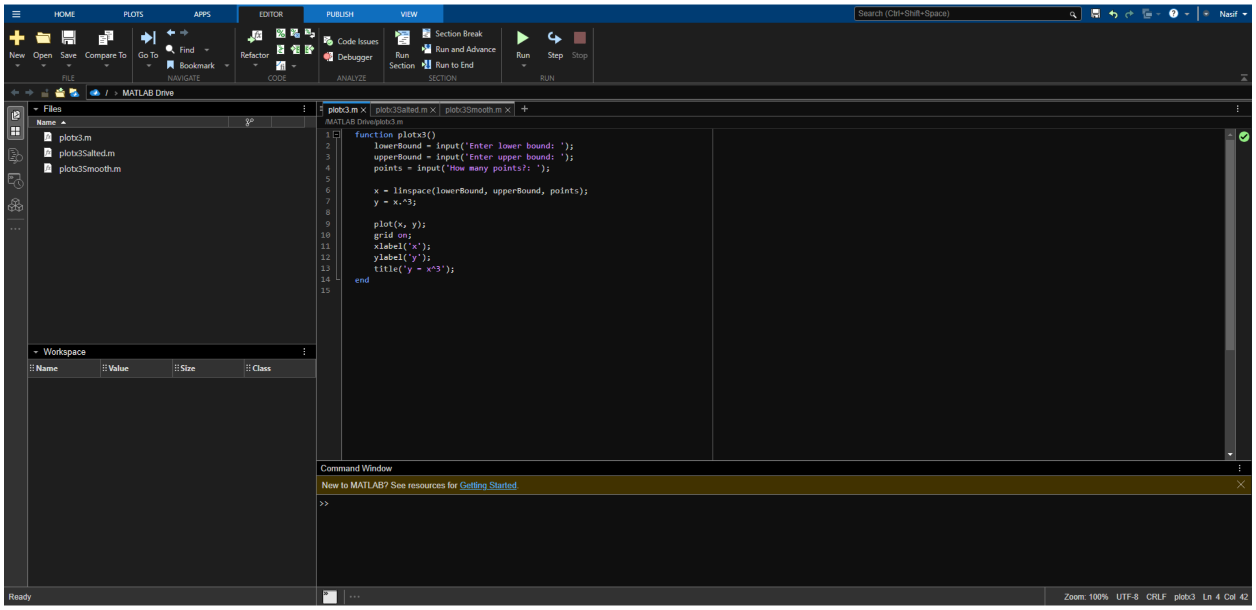


[-100, 100]
Interval: 0.1
Salt Range: 250000
Window: 1
Smoothing Count: 100

Just to experiment, I smoothed the same plot as previously with a window of 1 but doing so 100 times to see how the data would appear. Surprisingly, it's not an overly curved wave as we previously saw with 100 loops. Instead, there are now lots of smaller "heaps" along the form, but ignoring those, we can see that the original form of the function is recovered quite well.

PSS 2 - Matlab

For this section, I did not feel the need to repeat what I've done for my own Java implementation, where I tested various parameters and saw how it affected the data. As the Matlab implementation does the exact same thing as my Java implementation, I would needlessly be parroting previous information. Instead, I will go over how I implemented PSS using Matlab.



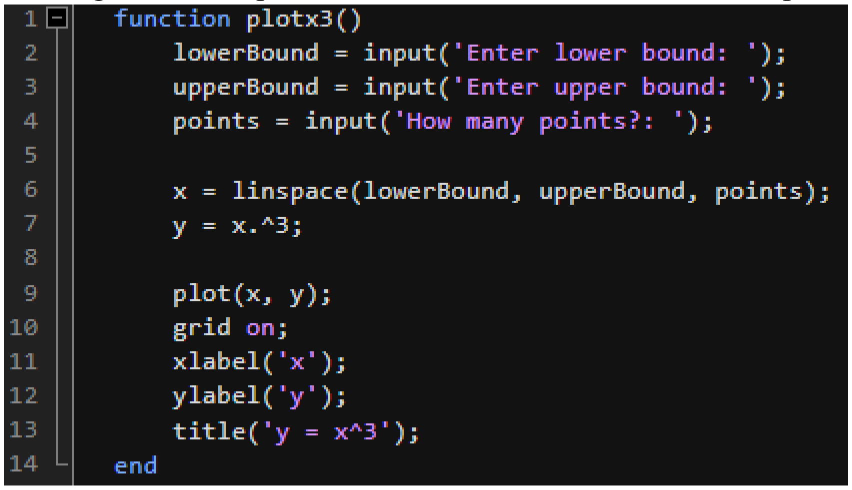
```
function plotx3()
    lowerBound = input('Enter lower bound: ');
    upperBound = input('Enter upper bound: ');
    points = input('How many points?: ');

    x = linspace(lowerBound, upperBound, points);
    y = x.^3;

    plot(x, y);
    grid on;
    xlabel('x');
    ylabel('y');
    title('y = x^3');
end
```

Fortunately, Matlab provides a free version of their application on their website, which I do *not* know whether it is time-limited or not simply because I'm too good at coding to find out. Regardless, the implementation for PSS was a lot more straightforward thanks to the built-in math functions and plot. In comparison to the 5 classes I wrote in Java, I've only needed 3 classes, the 3 of which were basically modifications of each other:

- One to plot a function, cleanly
- One to plot a function, salted, but also the original
- One to plot a function, smoothed, but also the previous two at once for comparison

Plotting

The code is short enough for me to paste into the document, which makes explaining simpler.



```
function plotx3()
    lowerBound = input('Enter lower bound: ');
    upperBound = input('Enter upper bound: ');
    points = input('How many points?: ');

    x = linspace(lowerBound, upperBound, points);
    y = x.^3;

    plot(x, y);
    grid on;
    xlabel('x');
    ylabel('y');
    title('y = x^3');
end
```

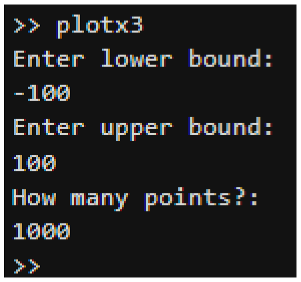
Functions in Matlab are bounded with the function keyword, the name, and the end keyword, highlighted in blue. Unlike Java, where one would need to import and instantiate a Scanner class, there's a built in "input()" method that prints the text parameter into the console and waits for a user response. I use this in all the programs for easy testing, since going into my Java implementation and tweaking the parameters in code for each test was getting a little tiresome.

"`x = linspace(a, b, n);`" makes a "row vector" of *n* evenly spaced points between the bounds *a* and *b*. This row vector holds data to plot later.

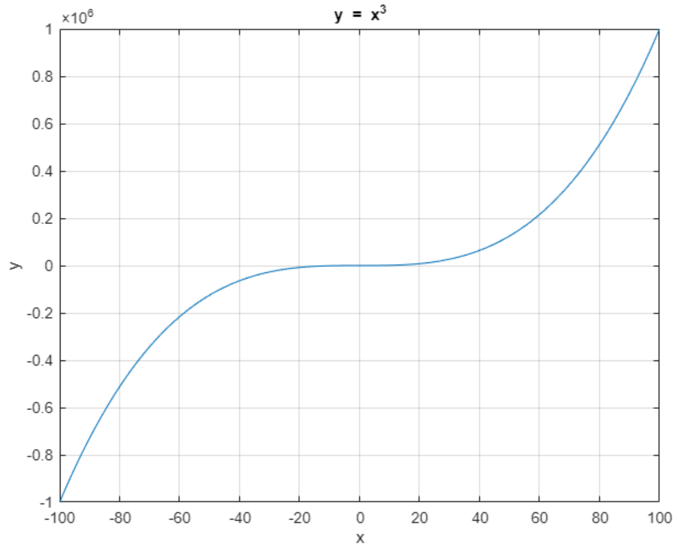
"`y = x.^3;`" holds our function to use.

"plot(x, y);" uses the row vector, *x*, and plots them following the formula of *y*. This opens up a new window where you can see the results of the plot. Following that, there's some statements to make the plot more appealing. I turn on a grid, I label the x-axis as "x", same for y, and I even give the graph a title.

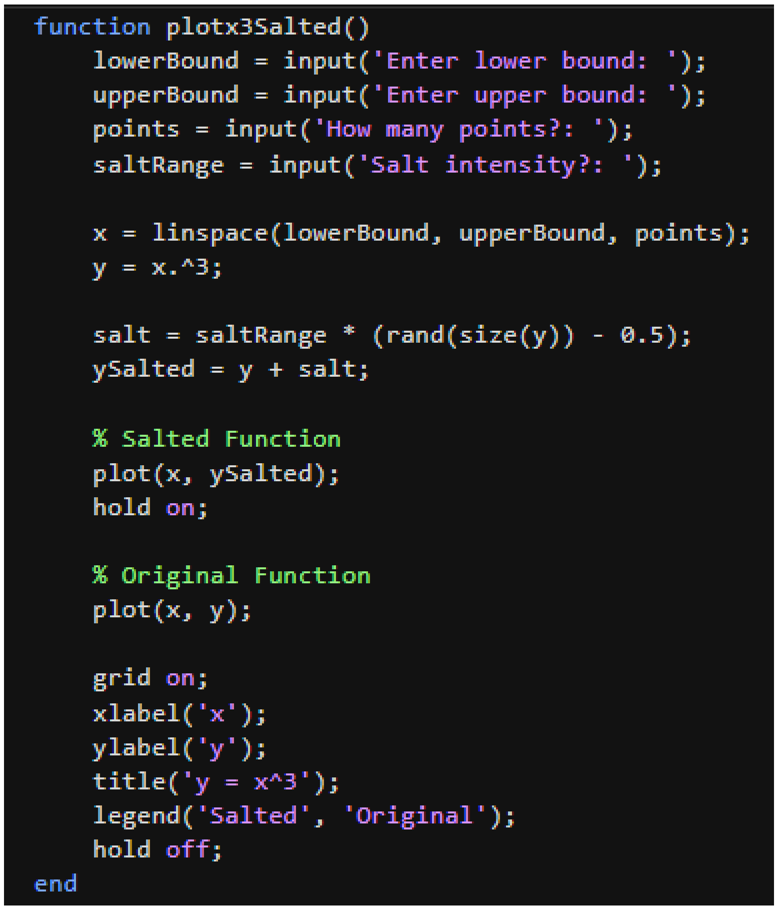
Here's the console as well as the plot that follows when running this program:



```
>> plotx3
Enter lower bound:
-100
Enter upper bound:
100
How many points?:
1000
>>
```

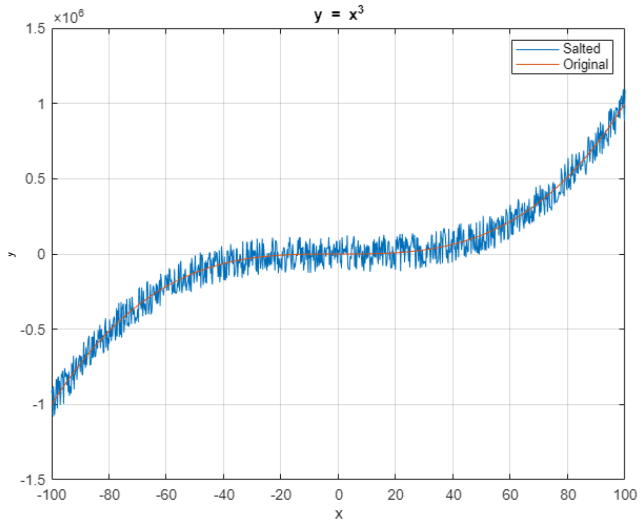


Salting

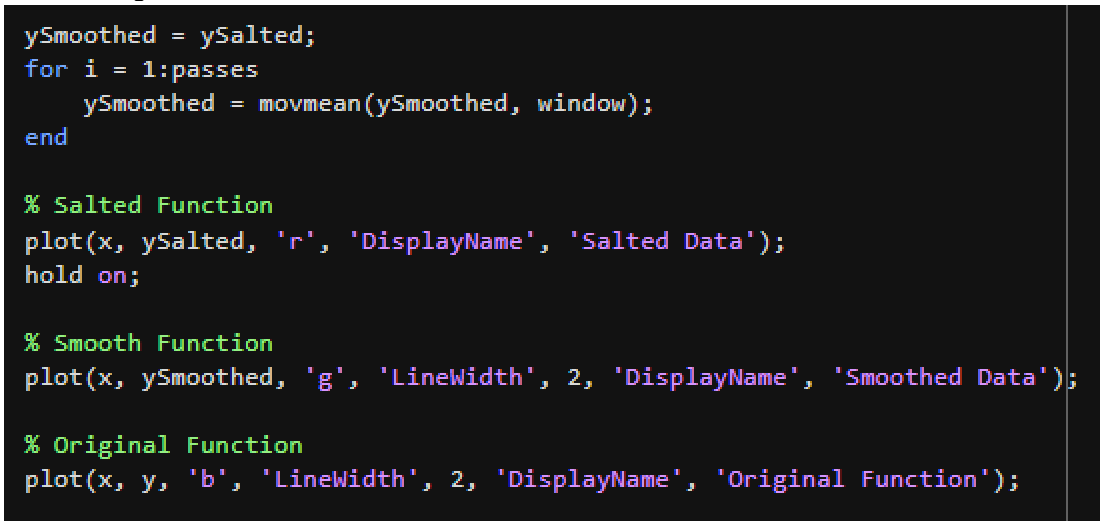


```
function plotx3Salted()
    lowerBound = input('Enter lower bound: ');
    upperBound = input('Enter upper bound: ');
    points = input('How many points?: ');
    saltRange = input('Salt intensity?: ');

    x = linspace(lowerBound, upperBound, points);
    y = x.^3;

    salt = saltRange * (rand(size(y)) - 0.5);
    ySalted = y + salt;

    % Salted Function
    plot(x, ySalted);
    hold on;

    % Original Function
    plot(x, y);

    grid on;
    xlabel('x');
    ylabel('y');
    title('y = x^3');
    legend('Salted', 'Original');
    hold off;
end
```

I need only make slight changes to the previous code to implement salting. I now ask for a salting range, which is used to calculate “salt”, which will salt y when added to it. We then plot the new data with the salted y-values. However, to also graph the original function, I enable “hold”, which disables the plot being cleared when plotting a new graph, then I plot the original. I do the same formatting as before and add a legend, making sure to disable hold since the option stays persistent across runs. “%” functions as a comment to better show which plot is which.

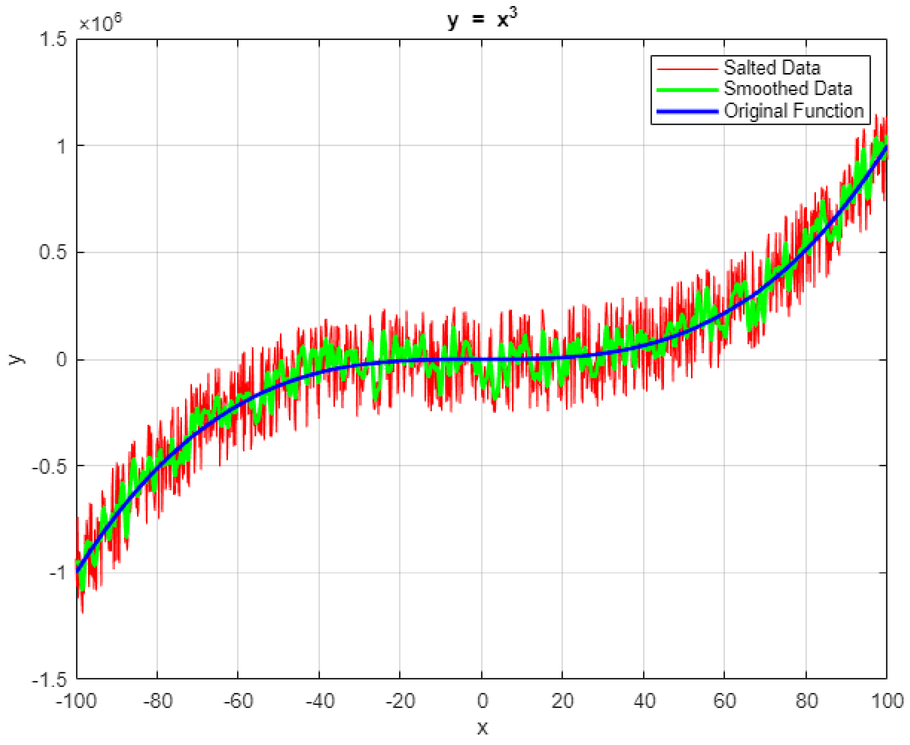


Smoothing



```
ySmoothed = ySalted;
for i = 1:passes
    ySmoothed = movmean(ySmoothed, window);
end

% Salted Function
plot(x, ySalted, 'r', 'DisplayName', 'Salted Data');
hold on;

% Smooth Function
plot(x, ySmoothed, 'g', 'LineWidth', 2, 'DisplayName', 'Smoothed Data');

% Original Function
plot(x, y, 'b', 'LineWidth', 2, 'DisplayName', 'Original Function');
```

The “movmean()” function acts perfectly to average values using a window in order to smooth out functions. I use a for-loop to do it multiple times. I also try a different method of plotting. ‘r’, ‘g’, and ‘b’ are simply colors for the line. If I put in a parameter ‘LineWidth’, I can follow it with an integer for the line width. Likewise, if I put in a parameter ‘DisplayName’, I can put another parameter of a string for the name. Finally, I get a beautifully formatted graph.


y = x^3
×10^6
y
x
Salted Data
Smoothed Data
Original Function