PSS2 REPORT

Intro to Octave

Arsalan Saleem

<u>Octave</u>

For this project I decided to use Octave, due to it being free and its availability in Linux repositories. It can also be installed easily with just one command:

```
sudo apt-get install octave
```

GNU Octave is a high-level interpreted language that is used for data analysis, mathematical computations, linear algebra, and much more. It can be used to easily solve differential equations, perform matrix computations, and visualize data. Octave is an open-source software that serves as a free alternative to MATLAB. It's designed for people who need a tool for numerical computing but may not have access to MATLAB due to its high cost or licensing restrictions. Octave is particularly useful because it closely follows MATLAB's syntax, making it easy for users familiar with MATLAB to transition. Whether you're a student learning math, a researcher analyzing data, or an engineer working on algorithms, Octave has something for everyone. It provides many built-in functions for tasks like matrix calculations, plotting graphs, solving equations, and working with data.

The tutorial I went over, (https://www.tutorialspoint.com/matlab/matlab_gnu_octave.htm), helped cover the basics of Octave, including how to enter commands, work with different variables, and use its different built-in functions. There are also many different videos online which help easily explain how to use Octave's features and utilize it to compute data along with different mathematical equations. The simple syntax being applied makes it very easy to learn and use for anyone looking to get started in this field. The tutorial also included basic programming principles such as conditional statements, loops and variables. The biggest strength of this software would be to plot and graph.

Just like any programming language it has data types and variables. Among supported data types are integers, doubles, complex numbers, strings, booleans, structures, and arrays.

Octave has built-in functions that I used in my code:

- **plot**: is used to plot the 2D data.
- **exists**: checks whether the given directory exists
- **title**: adds title/description above the plot
- **saveas**: saves the current window to the given file

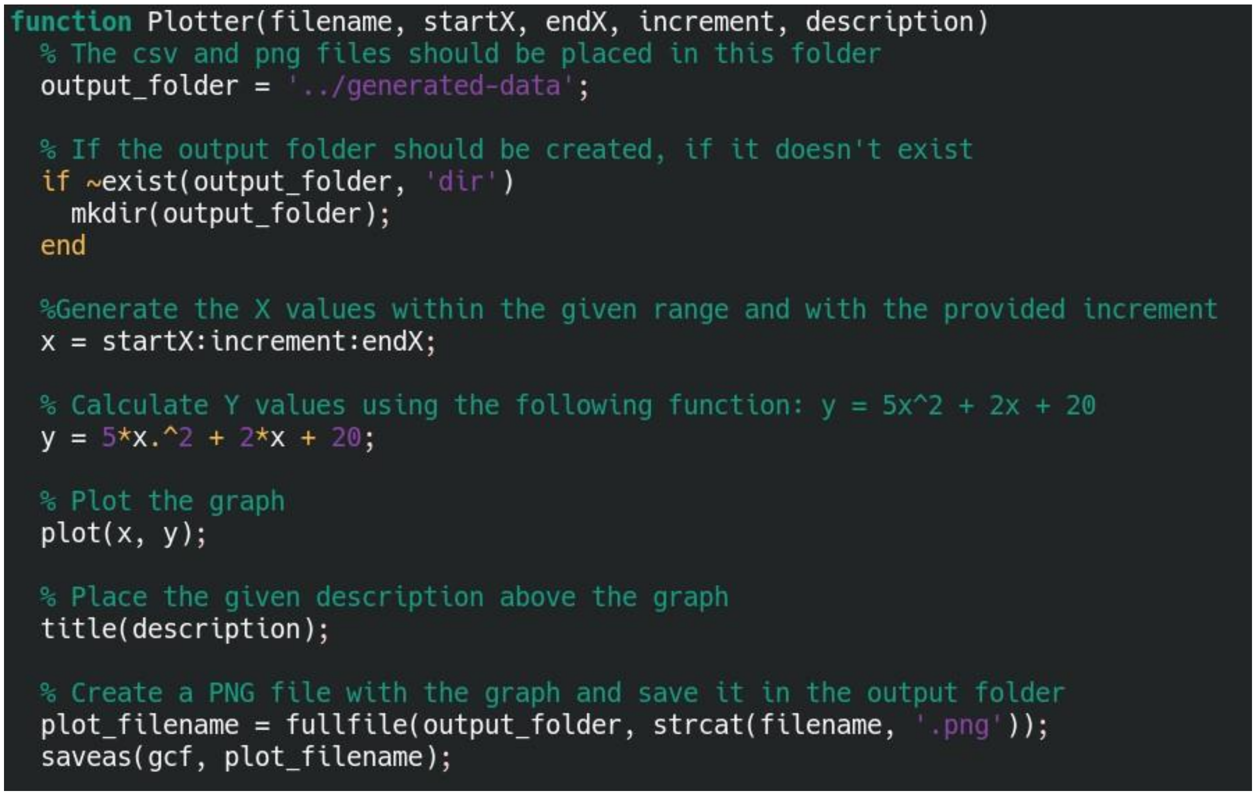
Plotter

The function I’ve chosen to plot using Octave is the following one:

$$y = 5x^2 + 2x + 20$$

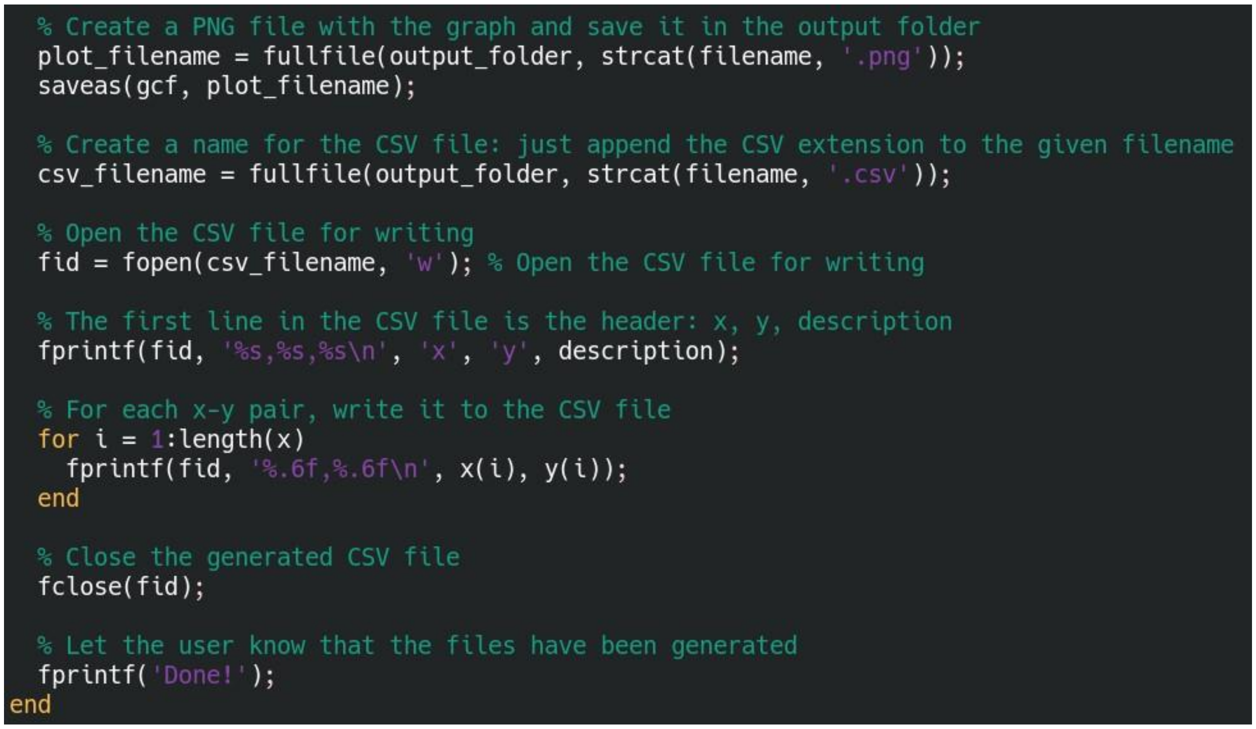
I modeled my **Plotter** function in Octave after my **Plotter** class in Java. The function should generate the x and y values for the chosen function, write the values in a **csv** file, and also generate a plot and save it in a **png** file. All generated files should be placed inside the **generated-data** directory. The function, just like the **generate** method in my Java class, should accept the name of the file (however, in this case, the name of the file should not contain an extension, the extension - **csv** or **png** is appended to the filename by my Octave code), the range and increment for x values, and the description that should be placed above the generated plot and in the third column of the first row in the **csv** file.

My **Plotter.m** code looks as follows:



```
function Plotter(filename, startX, endX, increment, description)
  % The csv and png files should be placed in this folder
  output_folder = '../generated-data';

  % If the output folder should be created, if it doesn't exist
  if ~exist(output_folder, 'dir')
    mkdir(output_folder);
  end

  %Generate the X values within the given range and with the provided increment
  x = startX:increment:endX;

  % Calculate Y values using the following function: y = 5x^2 + 2x + 20
  y = 5*x.^2 + 2*x + 20;

  % Plot the graph
  plot(x, y);

  % Place the given description above the graph
  title(description);

  % Create a PNG file with the graph and save it in the output folder
  plot_filename = fullfile(output_folder, strcat(filename, '.png'));
  saveas(gcf, plot_filename);
```

The x and y values are generated, plotted, and the plot is saved in a **png** file. The element-wise exponentiation operator **.^** is used, it operates on the vector of x values and generates a vector of y values.

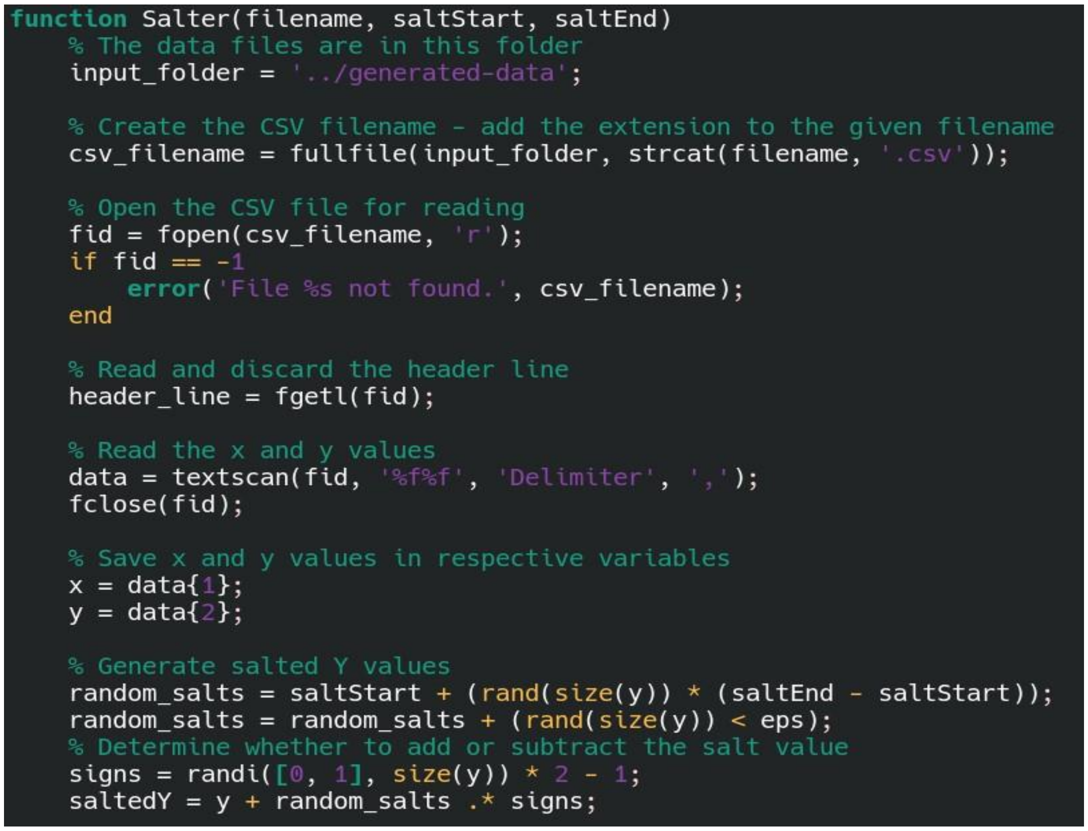


```
  % Create a PNG file with the graph and save it in the output folder
  plot_filename = fullfile(output_folder, strcat(filename, '.png'));
  saveas(gcf, plot_filename);

  % Create a name for the CSV file: just append the CSV extension to the given filename
  csv_filename = fullfile(output_folder, strcat(filename, '.csv'));

  % Open the CSV file for writing
  fid = fopen(csv_filename, 'w'); % Open the CSV file for writing

  % The first line in the CSV file is the header: x, y, description
  fprintf(fid, '%s,%s,%s\n', 'x', 'y', description);

  % For each x-y pair, write it to the CSV file
  for i = 1:length(x)
    fprintf(fid, '%.6f,%.6f\n', x(i), y(i));
  end

  % Close the generated CSV file
  fclose(fid);

  % Let the user know that the files have been generated
  fprintf('Done!');
end
```

Then the **csv** file is created, and the header and x and y values are written in it.

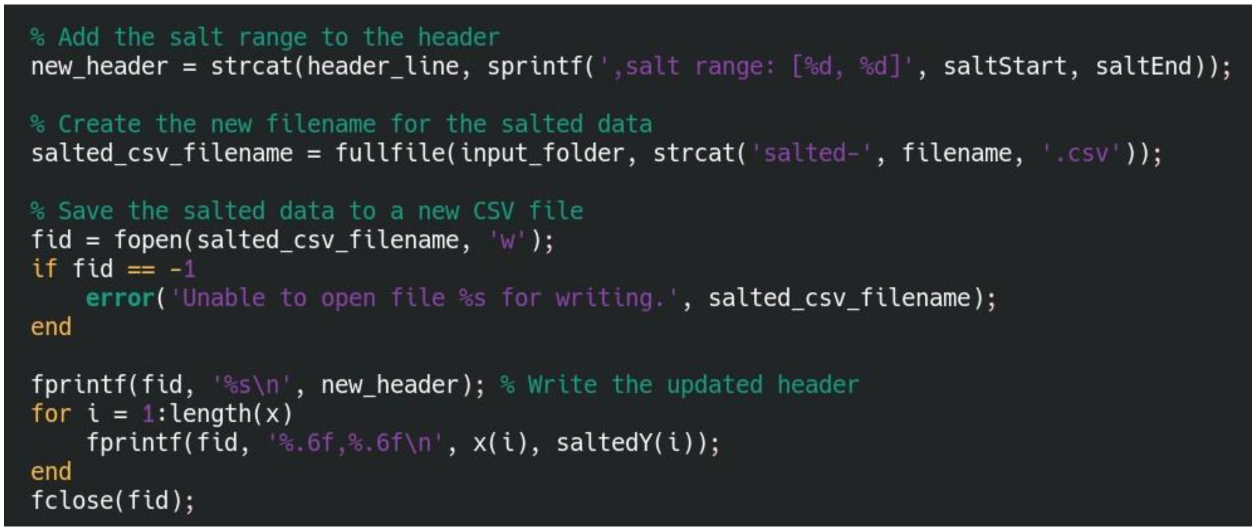
Salter

The **Salter** function mimics the logic of the **salt** method in the **Salter** Java class. It first reads the data from the **csv** file, then uses it to generate salted y values:



```
function Salter(filename, saltStart, saltEnd)
    % The data files are in this folder
    input_folder = '../generated-data';

    % Create the CSV filename - add the extension to the given filename
    csv_filename = fullfile(input_folder, strcat(filename, '.csv'));

    % Open the CSV file for reading
    fid = fopen(csv_filename, 'r');
    if fid == -1
        error('File %s not found.', csv_filename);
    end

    % Read and discard the header line
    header_line = fgetl(fid);

    % Read the x and y values
    data = textscan(fid, '%f%f', 'Delimiter', ',');
    fclose(fid);

    % Save x and y values in respective variables
    x = data{1};
    y = data{2};

    % Generate salted Y values
    random_salts = saltStart + (rand(size(y)) * (saltEnd - saltStart));
    random_salts = random_salts + (rand(size(y)) < eps);
    % Determine whether to add or subtract the salt value
    signs = randi([0, 1], size(y)) * 2 - 1;
    saltedY = y + random_salts .* signs;
```

After that the salted data is saved in a new **csv** file:



```
% Add the salt range to the header
new_header = strcat(header_line, sprintf(',salt range: [%d, %d]', saltStart, saltEnd));

% Create the new filename for the salted data
salted_csv_filename = fullfile(input_folder, strcat('salted-', filename, '.csv'));

% Save the salted data to a new CSV file
fid = fopen(salted_csv_filename, 'w');
if fid == -1
    error('Unable to open file %s for writing.', salted_csv_filename);
end

fprintf(fid, '%s\n', new_header); % Write the updated header
for i = 1:length(x)
    fprintf(fid, '%.6f,%.6f\n', x(i), saltedY(i));
end
fclose(fid);
```

And then the salted data is plotted and saved in a **png** file:

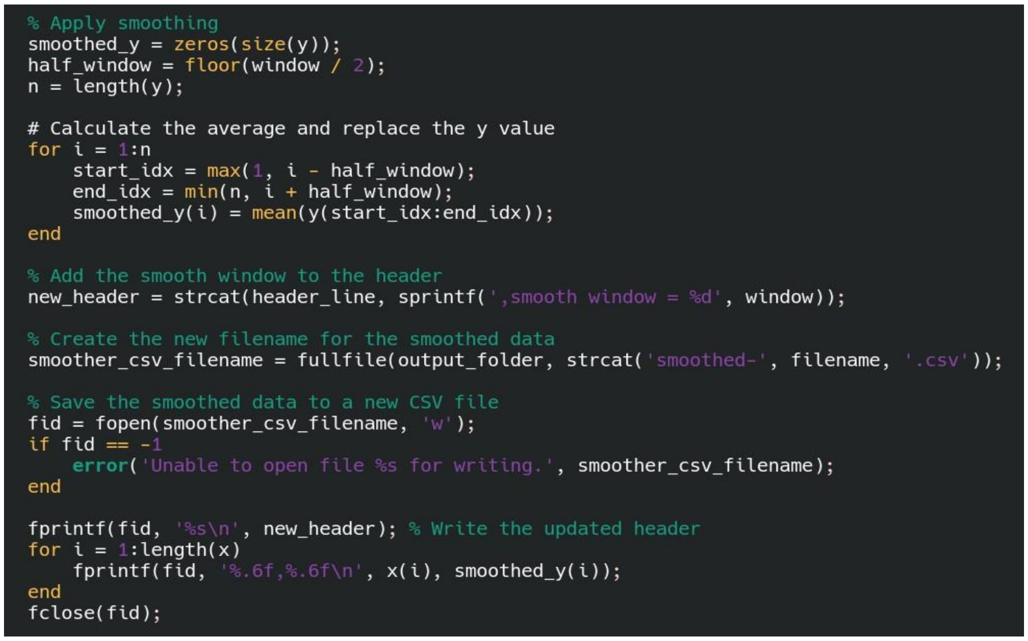
```
    % Plot the salted data
    figure;
    plot(x, saltedY, 'r'); % Red line for salted data
    title(sprintf('Salted Data for %s.csv (Range: [%d, %d])', filename, saltStart, saltEnd));

    xlabel('x');
    ylabel('Salted y');

    % Save the plot as a PNG file
    salted_plot_filename = fullfile(input_folder, strcat('salted-', filename, '.png'));
    saveas(gcf, salted_plot_filename);

    % Let the user done the salted data and plot have been generated
    fprintf('Done!');
end
```

Smoother

The **Smoother** function accepts the filename (without an extension) and the window value. Just like in the Java code, it reads the data from the file, then calculates the average of the y values within the given window:

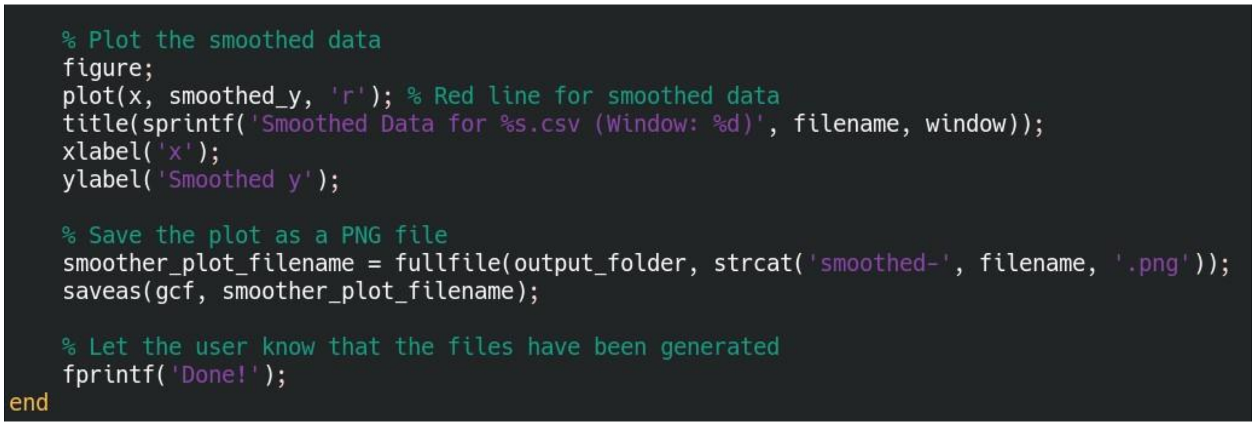
```
function Smoother(filename, window)
    % The data files are located in this folder
    input_folder = '../generated-data';
    output_folder = '../generated-data';

    % Create the CSV filename - add the extension to the given filename
    csv_filename = fullfile(input_folder, strcat(filename, '.csv'));

    % Open the CSV file for reading
    fid = fopen(csv_filename, 'r');
    if fid == -1
        error('File %s not found.', csv_filename);
    end

    % Read and discard the header line
    header_line = fgetl(fid);

    % Read the x and y values
    data = textscan(fid, '%f%f', 'Delimiter', ',');
    fclose(fid);

    % Save x and y values in respective variables
    x = data{1};
    y = data{2};

    % Check for sufficient data
    if length(y) < window
        error("Insufficient data!");
    end
```

And creates smoothed y values. The smoothed data is then saved in a new **csv** file:



```
% Apply smoothing
smoothed_y = zeros(size(y));
half_window = floor(window / 2);
n = length(y);

# Calculate the average and replace the y value
for i = 1:n
    start_idx = max(1, i - half_window);
    end_idx = min(n, i + half_window);
    smoothed_y(i) = mean(y(start_idx:end_idx));
end

% Add the smooth window to the header
new_header = strcat(header_line, sprintf(',smooth window = %d', window));

% Create the new filename for the smoothed data
smoother_csv_filename = fullfile(output_folder, strcat('smoothed-', filename, '.csv'));

% Save the smoothed data to a new CSV file
fid = fopen(smoother_csv_filename, 'w');
if fid == -1
    error('Unable to open file %s for writing.', smoother_csv_filename);
end

fprintf(fid, '%s\n', new_header); % Write the updated header
for i = 1:length(x)
    fprintf(fid, '%.6f,%.6f\n', x(i), smoothed_y(i));
end
fclose(fid);
```

Then the smoothed data is plotted and saved as a **png** file:

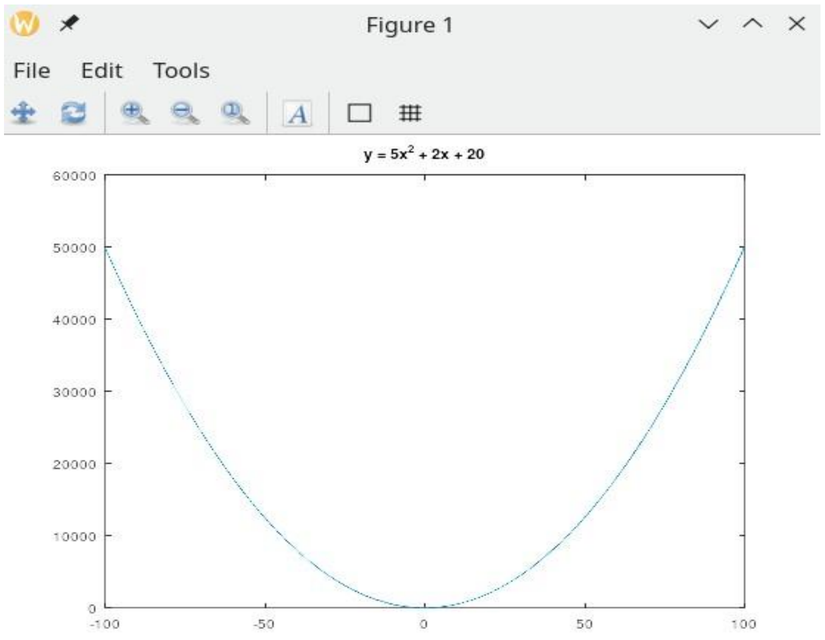


```
    % Plot the smoothed data
    figure;
    plot(x, smoothed_y, 'r'); % Red line for smoothed data
    title(sprintf('Smoothed Data for %s.csv (Window: %d)', filename, window));
    xlabel('x');
    ylabel('Smoothed y');

    % Save the plot as a PNG file
    smoother_plot_filename = fullfile(output_folder, strcat('smoothed-', filename, '.png'));
    saveas(gcf, smoother_plot_filename);

    % Let the user know that the files have been generated
    fprintf('Done!');
end
```

I call each of the described functions, one after another as follows:

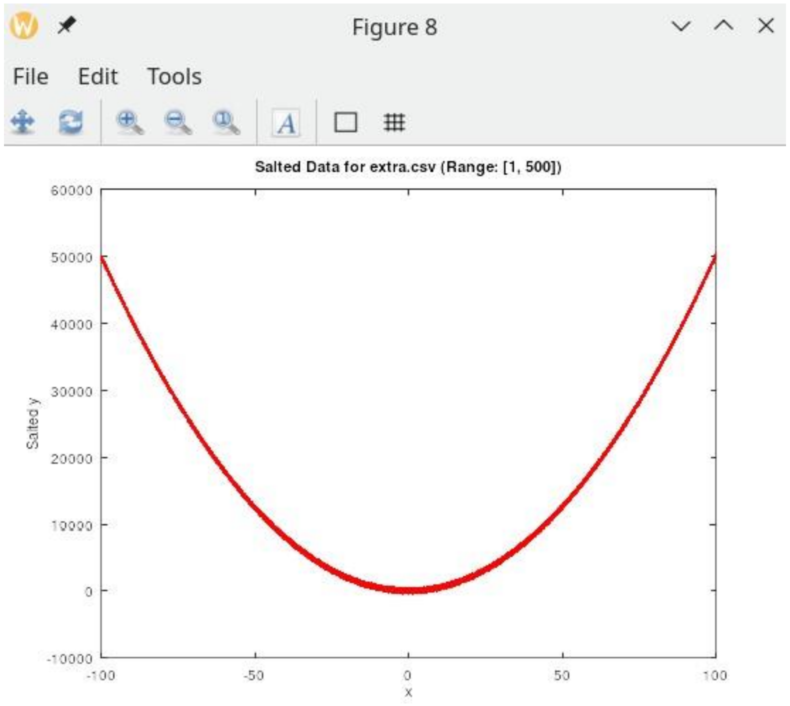
Plotting the original data

```
Plotter('extra', -100, 100, 0.01, 'y = 5x^2 + 2x + 20');
```



Plotting the salted data

```
Salter('extra', 1, 500);
```



Plotting the smoothed salted data

```
Smoother('salted-extra', 10);
```

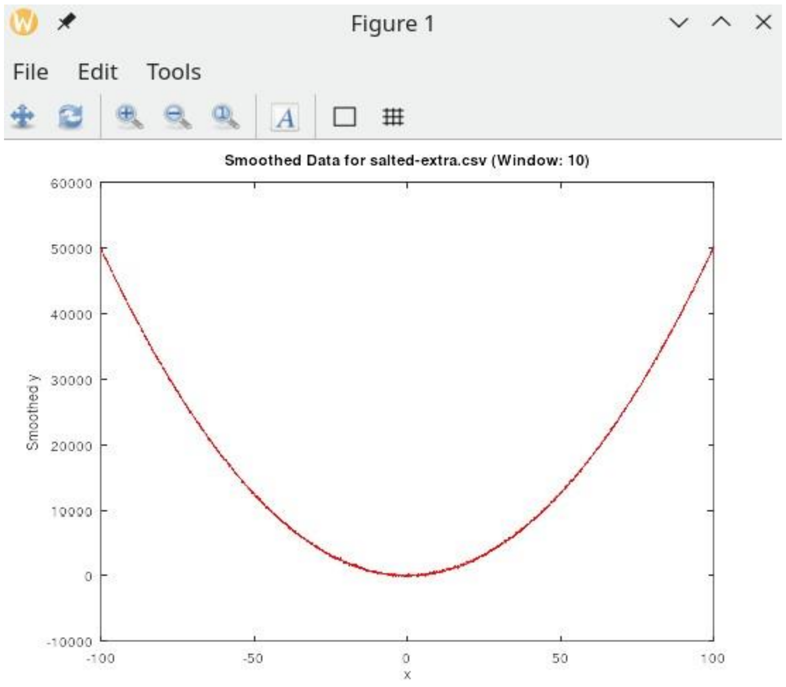
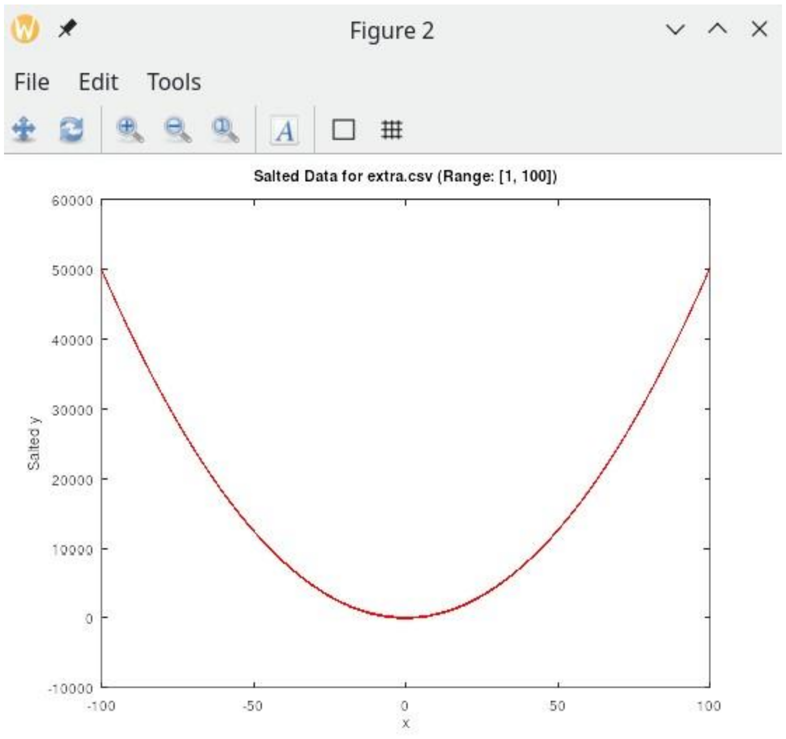
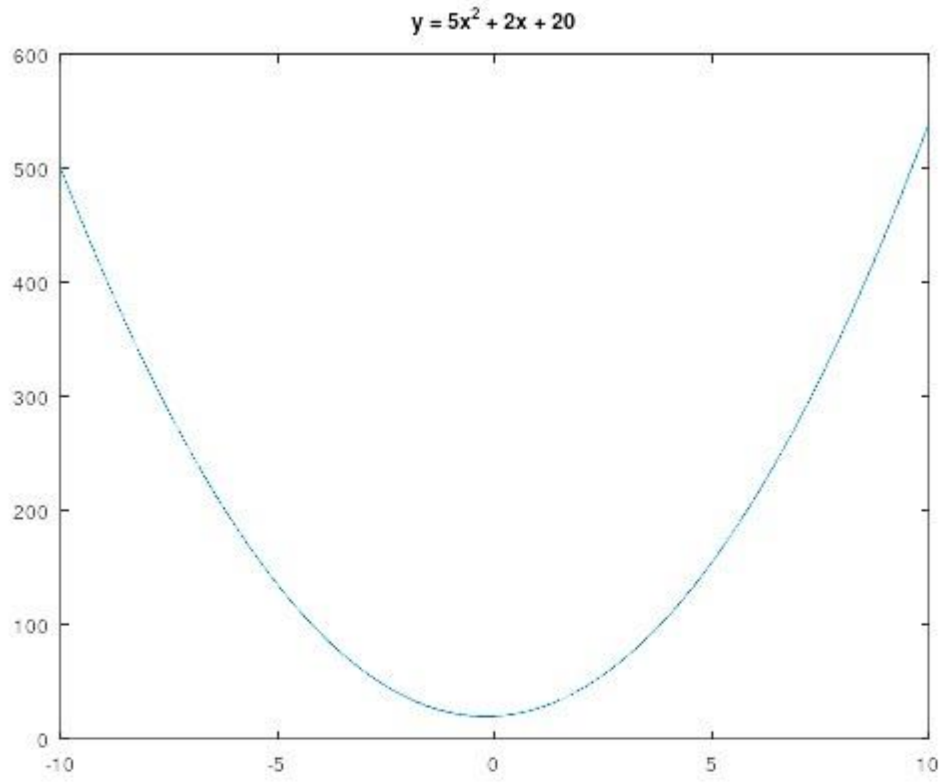


Figure 1
File
Edit
Tools
Smoothed Data for salted-extra.csv (Window: 10)
Smoothed y
x
60000
50000
40000
30000
20000
10000
0
-10000
-100
-50
0
50
100

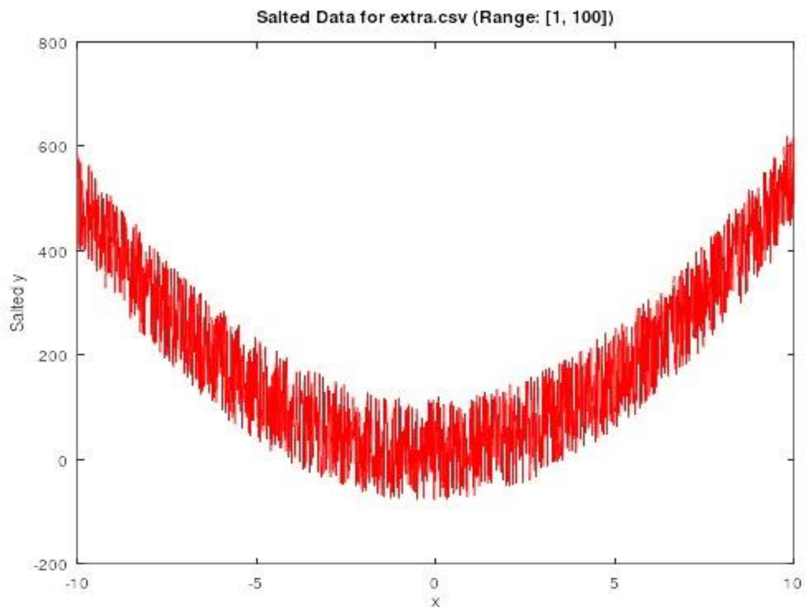


The effect of salting is easier to see if we zoom in, by using the smaller range of x values:

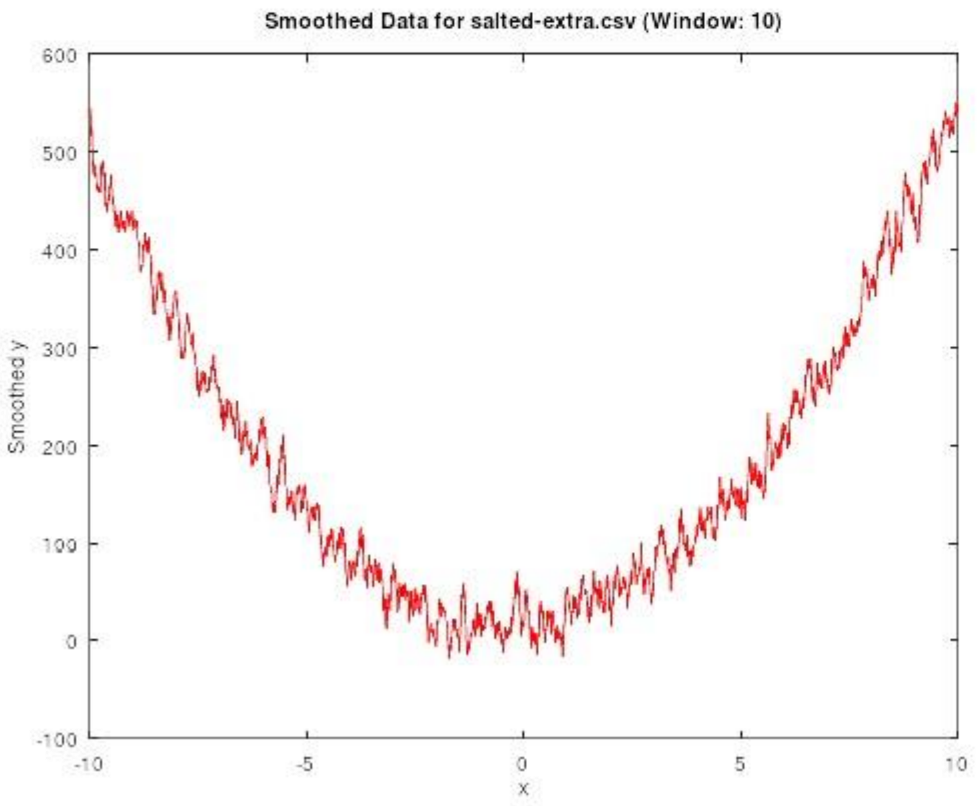
```
Plotter('extra', -10, 10, 0.01, 'y = 5x^2 + 2x + 20');
```



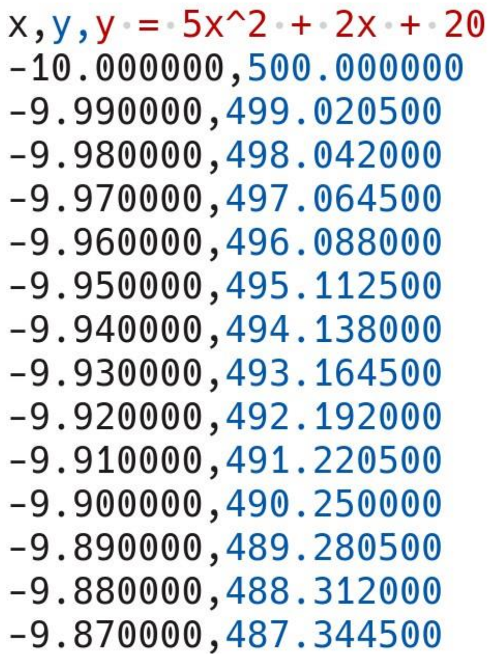
```
Salter('extra', 1, 100);
```



```
Smoother('salted-extra', 10);
```

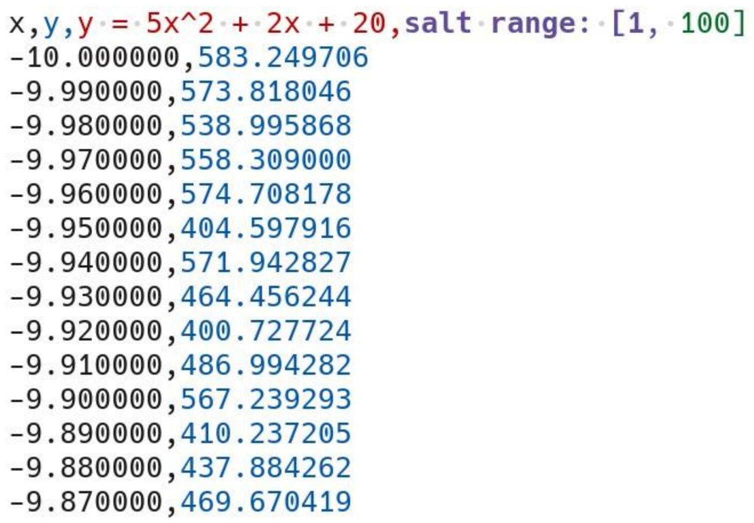


Then the **extra.csv** starts with these rows:



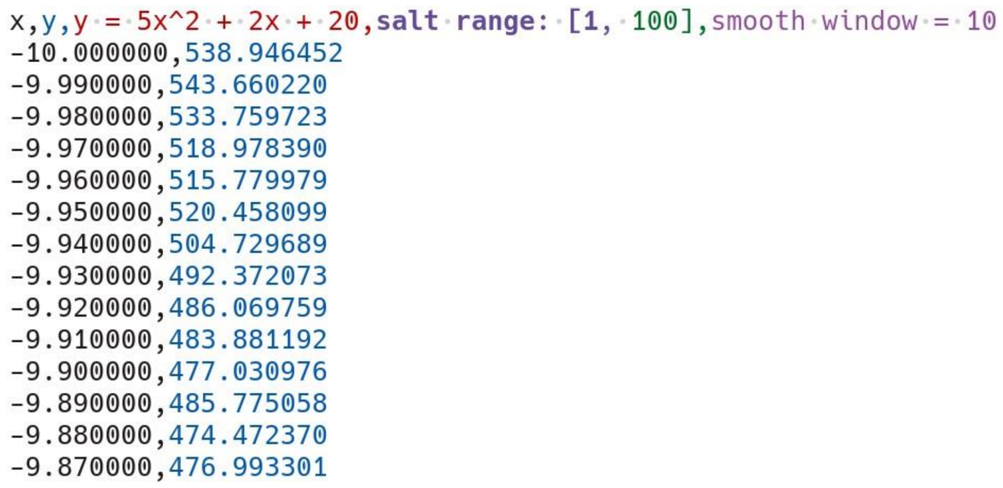
```
x,y,y = 5x^2 + 2x + 20
-10.000000,500.000000
-9.990000,499.020500
-9.980000,498.042000
-9.970000,497.064500
-9.960000,496.088000
-9.950000,495.112500
-9.940000,494.138000
-9.930000,493.164500
-9.920000,492.192000
-9.910000,491.220500
-9.900000,490.250000
-9.890000,489.280500
-9.880000,488.312000
-9.870000,487.344500
```

The **salted-extra.csv**:



```
x,y,y = 5x^2 + 2x + 20,salt range: [1, 100]
-10.000000,583.249706
-9.990000,573.818046
-9.980000,538.995868
-9.970000,558.309000
-9.960000,574.708178
-9.950000,404.597916
-9.940000,571.942827
-9.930000,464.456244
-9.920000,400.727724
-9.910000,486.994282
-9.900000,567.239293
-9.890000,410.237205
-9.880000,437.884262
-9.870000,469.670419
```

And the **smoothed-salted-extra.csv**:



```
x,y,y = 5x^2 + 2x + 20,salt range: [1, 100],smooth window = 10
-10.000000,538.946452
-9.990000,543.660220
-9.980000,533.759723
-9.970000,518.978390
-9.960000,515.779979
-9.950000,520.458099
-9.940000,504.729689
-9.930000,492.372073
-9.920000,486.069759
-9.910000,483.881192
-9.900000,477.030976
-9.890000,485.775058
-9.880000,474.472370
-9.870000,476.993301
```

We can test how different salt ranges and smooth windows change the output.

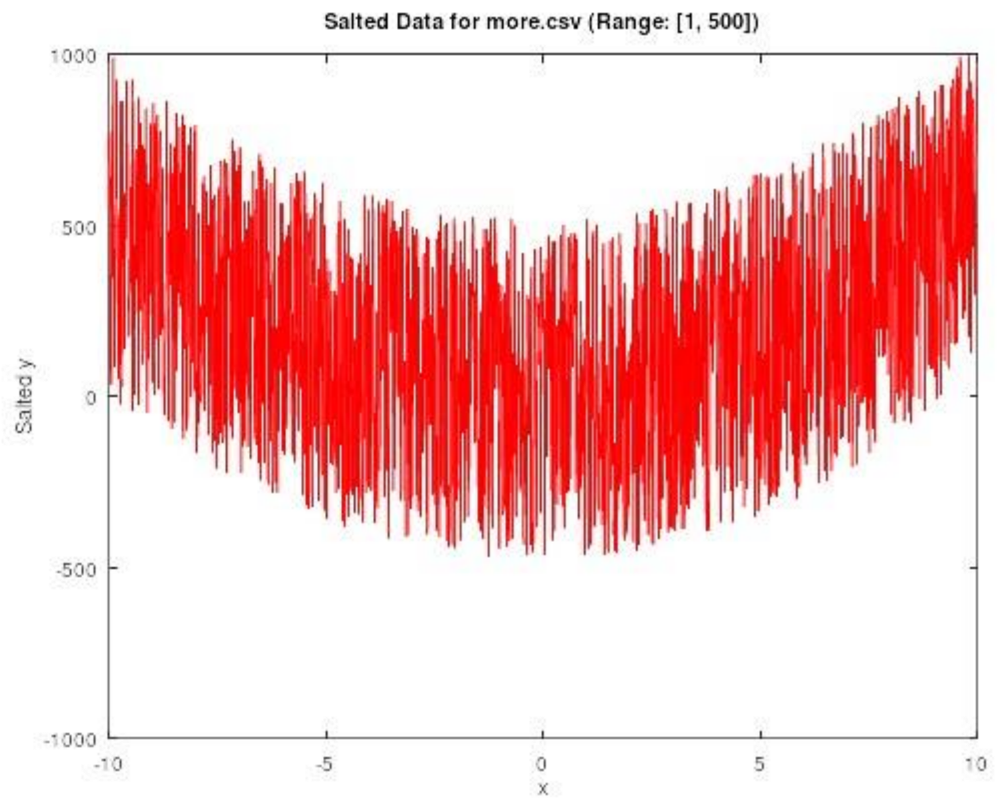
First let's create a new file, so that the **extra.csv** is not overwritten:

```
Plotter('more', -10, 10, 0.01, 'y = 5x^2 + 2x + 20');
```

Then make the salt range bigger:

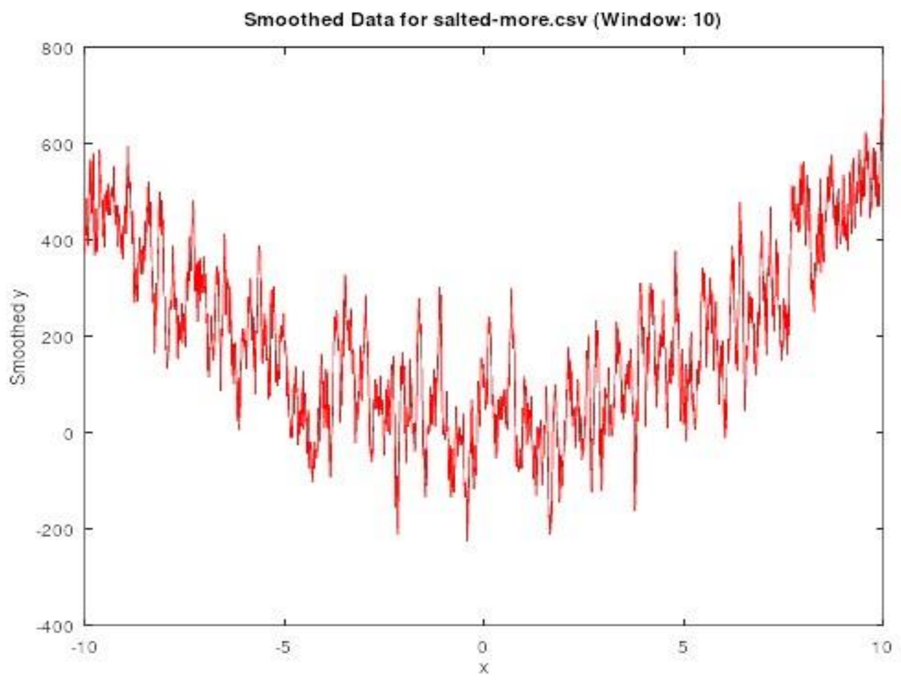
```
Salter('more', 1, 500);
```

The salted output:

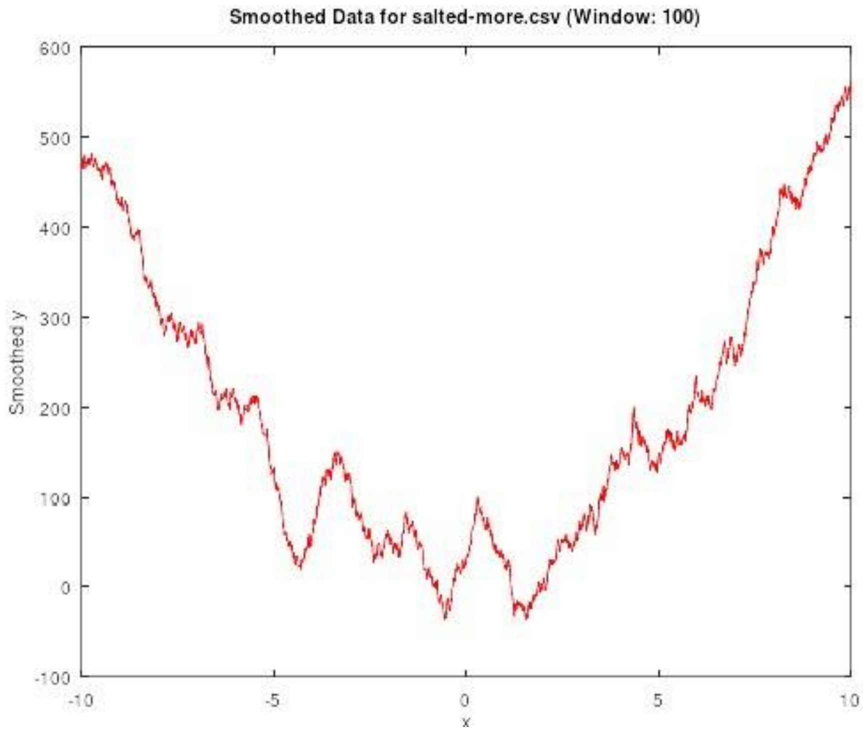


Let’s smooth it:

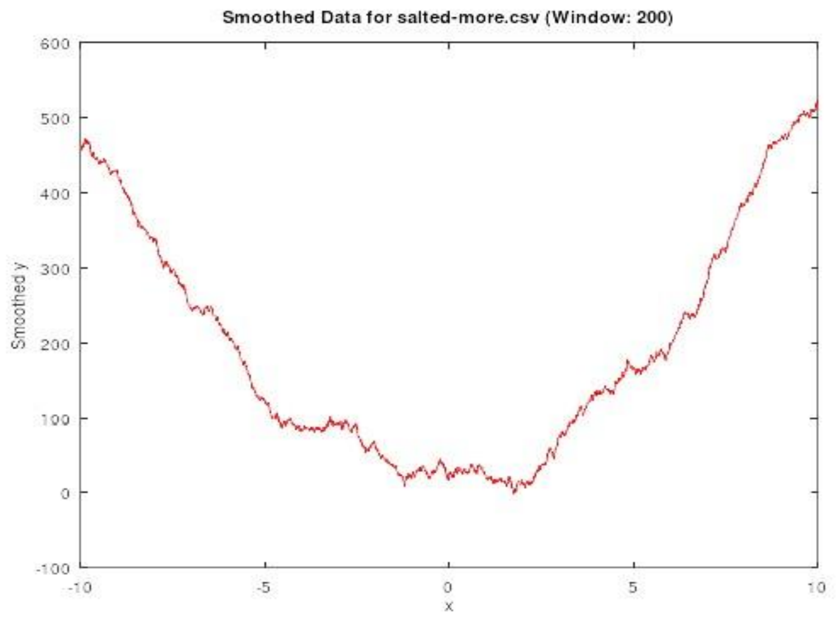
```
Smoother('salted-more', 10);
```



```
Smoother('salted-more', 100);
```



```
Smoother('salted-more', 200);
```



It’s easy to see how the larger salt range results in a greater variance in the y values, and the larger window value results in a more smoothed out graph.

Conclusion

This program’s main goal was to produce data correlating to a function, which was y = 5x^2 + 2x + 20 in this case. After generating the dataset, we added random noise to the data using the process of Salting. Once this step is complete, it is vital to smooth the data in order to reduce the noise and make the data more recognizable. Finally, one is able to plot the data after completing these steps. The plotter method generated the data. Different sets of trials were conducted with smaller samples and larger samples in order to visualize the noise in the graphs. It was easy to conclude how the larger salt range resulted in a greater variance in all of the y values, and the larger window value resulted in a more smoothed out graph.

Some ways to improve the program could be to add error handling, and to have flexible parameters inside the code. This would make the functions more adaptable, and we would be able to handle a wider range of datasets. Overall, being able to manipulate data and graphs in Octave is a very valuable asset, and the simplicity of the programming makes it easy to understand and enjoy. In the future, I will definitely be looking to add to my skillset in this language and explore the different functionalities Octave has to offer.