PSS User Manual

By: Jacob Attia

Introduction:

Project goal
Create a small, language-agnostic data pipeline that lets anyone

1. generate a clean set of (x,y) points for an arbitrary nonlinear function (the Plotter),

2. inject controlled random noise into those points (the Salter), and

3. remove that noise by a moving-average filter (the Smoother),
   while exporting every intermediate result to CSV and/or showing the curve on-screen.

To demonstrate that the ideas are portable, the same three steps were implemented in **three toolchains**:

BlueJ (Java)
- Writes the CSVs only, the user opens them in Excel and inserts a scatter chart manually.

GNU Octave
- Writes the CSVs and pops up a figure window automatically.

VS Code (Java + JFreeChart)
- Writes the CSVs and shows a Swing chart window via JFreeChart.

BlueJ Program:

The Java / BlueJ version keeps the workflow purposely console-only so that the user can decide later whether to view the data in Excel, Octave, or any other plotting tool. It is organised into four small classes, each with a single responsibility and a very short "main" method.

PlotFunction.java

- Generate pristine (x,y) pairs and write "plot_data.csv"
    - Separate the math "(myFunction)" from the I/O. A simple "for" loop over "x" fills a "List<double[ ]>", then one "try-with-resources" block streams the list to disk.

DataSalter.java

- Read the clean CSV, add uniform noise ±range to Y only, write "salted_data.csv"
    - Reuse the same "read XYFromCSV/writeXYToCSV" utility for symmetry, noise comes from a single "Random" instance so we can add a seed later if reproducibility is needed.

Smoother.java

- Read any CSV, replace each Y with the average of its neighbours in a ±window (default 5), write "smoothed_data.csv"
    - Compute into a temporary "smoothedY[ ]" array first, then copy back. This prevents the "cascade" effect you'd get if you overwrote each value in place.

DataHandler.java

- Convenience driver that calls the three main methods in sequence.
    - Keeps orchestration logic out of the individual classes so they remain reusable on their own

Coding Structure

- Pick one easy file format first
    - Goal: let every step hand data to the next one without fuss.
    - Choice: a two-column CSV called X,Y.
    - Why: Excel, Octave, and JFreeChart can all open the same file, so I only had to write one tiny save/load method.
- Write one small program per job
    - Plotter makes the clean points, Salter adds noise, Smoother removes noise, and DataHandler just runs them in order.

    - Keeping each job in its own file means I can test or change one piece without touching the others.
- Keep every loop obvious
    - Plotter: loop through x-values, compute y, save.
    - Salter: loop through the same list, add a random ±number to y.
    - Smoother: loop again, replace y with the average of its neighbours.
    - One pass per task—no fancy tricks—so the code stays short and easy to follow.
- Reuse the same read/write helpers everywhere
    - I wrote readCSV and writeCSV once, then called them in all three classes. Less code to maintain and fewer places for mistakes.

- Summary
    - In short: decide on a simple file format, split the work into tiny classes, write clear loops, and share helper methods. That approach kept the project small, readable, and easy to move later to Octave and JFreeChart.

Octave Program:

The Octave set (plot_function.m, salter.m, smoother.m) repeats the very same three-step data journey which generates -> add noise -> smooth, but this time the scripts draw the graph automatically as soon as you run them.

Plot_function.m

- Builds vectors x and $y = x^2$, calls “plot”, and writes “plot_data.csv”
    - Exact same math loop as “PlotFunction.java”, just vectorised

Salter.m

- Reads the CSV, creates uniform noise in “±range”, adds it to y, plots “original vs. salted”, saves “salted_data.csv”
    - Same formula as Java’s “DataSalter”, but three lines of Octave do the whole job thanks to native matrix ops

Smoother.m

- Loads any CSV, replaces each y with the average of neighbours inside “±window”, plots both curves, writes smoothed_data.csv
    - Logic identical to Java’s Smoother, loops kept for clarity even though Octave could vectorize the mean.

Coding Structure

- Reuse the CSV contract
  - Stuck with the same two-column layout so the files coming out of Java drop straight into Octave with “csvread”

- Let Octave draw everything for free
  - Each script ends with a “figure; plot(...)” block, so the moment you run plot_function, salter, or smoother, a window pops up—no manual Excel step this time

- Keep parameters at the top
  - Every script begins with easy-to-see defaults (x_start, range, window), but you can override them from the command line
  - for example, (salter(3,"plot_data.csv", "noisy.csv", 1234);

- Stay readable
  - Even though Octave can replace the smoothing loop with convolutions, the explicit “for” loop matches the Java logic one-for-one, making the code easy to compare across languages

- Summary
  - With that, the Octave layer works as a drop-in visual tester: run the script -> see the graph -> confirm the CSV, all without leaving the Octave console

VS code (JfreeChart Implementation + Apache stats library)

The third environment repeats the plot -> salt -> smooth flow, but this time every program opens its own Swing chart window the moment you run it. That live graph comes from the JFreeChart library, which you added to VS Code as a single .jar file in Referenced Libraries.

DataPlotter.java

- Generates clean (x,y), writes plot_data.csv, shows one curve
  - Builds an “XY” Series, wraps it in an “XY” SeriesCollection, then calls ChartFactory.createXYLineChart(...) and drops the result into a ChartPanel.

DataPlotter.java (added apache stats library)

- Result: when I run DataPlotter the window now shows the original parabola *plus* a flat reference line at the average yyy-value, and the console prints the mean and standard deviation.
- No other parts of the pipeline changed—adding Commons Math simply gives me extra numeric summaries (or, later, regressions and confidence bands) whenever I want them, while JFreeChart takes care of displaying the new information.

DataSalter.java

- Reads the clean CSV, adds uniform noise ±range, writes salted_data.csv, shows two curves (original vs. salted)
    - Creates two "XY" Series objects, adds them to the same dataset, sends that to createXYLineChart, so legend and colors appear automatically

DataSmoother.java

- Reads any CSV, applies moving-average window ±k, writes smoothed_data.csv, shows two curves (original vs. smoothed)
    - Same pattern as DataSalter; just a different label and dataset

Coding Structure

- **Add Jfreechart to Library**
    - Download jfreechart-1.5.2.jar -> put it in a lib/ folder -> click the "+" next to Referenced Libraries in VS Code and pick the jar. From that moment the import org.jfree.chart.* lines turn green

- **Turning a list of points into a chart**
    - First we create a series object and pour every (x, y) pair into it—think of the series as a column in a spreadsheet that knows its own label ("original"). Next we drop that series into a dataset container; this is what JFreeChart expects when it builds graphs. Finally we hand that dataset, plus a title and axis labels, to JFreeChart's factory method. In one call the library returns a finished chart object, already equipped with axes, grid lines, default colours, and legend support.

- **Showing the chart on screen**
    - The chart object is just data until we place it inside a Swing component. JFreeChart provides a ready-made ChartPanel, which is like a picture frame for the plot. We put that panel into a tiny JFrame, let the frame auto-size (pack), and make it visible. The result is a standalone window

that pops up as soon as the program runs, displaying the curve without any extra GUI coding on our part.

Conclusion

All three versions do the same job: make clean data, add noise, then smooth it out. In BlueJ we save the points to a CSV and graph them in Excel. In Octave we run a script and the graph pops up right away. In VS Code we run the program and JFreeChart shows the graph for us. Using the same two-column CSV in every step keeps the process easy to share and repeat.