InJunction: STM-BJ Data Processing Project Documentation

InJunction is an interactive data processing software for *Single-Molecule Break Junction (STM-BJ)* experiments. It provides a GUI built with PySide6 (Qt) to analyze molecular junction data, including conductance traces, I–V curves, and noise measurements. The project integrates a modern UI framework by Wanderson Pimenta (PyDracula theme) and adapts code from the Xiamen University **XMe_DataAnalysis** toolkit. It runs on Python 3.10 with scientific libraries (NumPy, SciPy, scikit-learn, etc.) as listed in **requirements.txt**.

Feature Overview: InJunction supports multiple analysis modes covering the typical workflow of STM-BJ data processing:

- **Conductance histogram generation** (1D and 2D) and peak fitting
- **Automated clustering of conductance traces** (with K-means, GMM, or time-series clustering)
- **Optimal cluster number determination** via Calinski–Harabasz (CH) index (“CH scan”)
- **Merging and saving of clusters** for further analysis
- **Segmentation (“subsection”) analysis** of traces over time or length
- **Fowler–Nordheim (FN) transform analysis** of I–V data
- **I–V curve processing** from raw .tdms files to cleaned .npz datasets
- **Capacitive artifact removal** (“de-capacity”) in I–V curves
- **Power spectral density (PSD) analysis** of conductance noise

Underlying these features are several Python modules and scripts. This document explains each component and how they interconnect in the data processing workflow.

Project Structure and GUI Architecture

Main Application (GUI): The GUI is defined in **main.ui** (a Qt Designer file) and loaded by **main.py**. The interface uses a sidebar menu (PyDracula theme) with buttons for different pages (Home, Cluster Analysis, Subsection Analysis, I–V Analysis, PSD, etc.), and a central stacked widget to display each functional page. The main window class (in main.py) initializes Ui_MainWindow from the UI file and connects UI events (button clicks, etc.) to corresponding Python functions. Key UI navigation buttons include:

- **Home** – Overview page (e.g., may show an introduction or quick links)
- **Cluster cacu** – Conductance clustering & histogram page
- **Subsection cacu** – Data segmentation and length analysis page
- **Find trace (FN transform)** – Single-trace analysis page (intended for FN transform)
- **IV analyse** – I–V processing page
- **PSD** – Noise/PSD analysis page

When the user clicks a sidebar button, the application switches the QStackedWidget to the respective page. The UI elements (buttons, sliders, text boxes) on each page are connected to processing functions. The modern GUI template provides window controls (minimize, maximize, close) and animated sidebar toggling. The **modules/** directory contains the UI helper modules from the PyDracula framework (for example, UiFunctions.toggleMenu for the sidebar) – these handle interface animations and theming.

Data Workflow Integration: The main.py script orchestrates user interactions. For instance, when an analysis is triggered (e.g., user selects files or clicks “Run”), main.py will call the appropriate function from the analysis modules (CaculateHistogram.py, new_function.py, ivDataProcessUtils.py, etc.), then collect the results and update the GUI (plots, text outputs).

Long computations (like I–V processing or clustering) are offloaded via threads or background calls to keep the UI responsive (e.g., using Qt signals/slots or simply updating after completion). The main script also manages reading/writing files through dialog interactions (e.g., open .tdms or .npz files, saving results) and uses pop-up messages for user feedback (via pymsgbox for simple alerts).

Conductance Histogram Generation and Clustering Analysis

One core functionality is analyzing large sets of conductance–distance traces from break-junction experiments. This is handled primarily in **CaculateHistogram.py** (which, despite the name, contains many analysis functions). Key features on the **"Cluster cacu"** page include: *conductance histograms*, *trace clustering*, and *optimal cluster count determination*.

Histogram Calculation (1D & 2D)

For a given dataset of conductance traces, the software can compute standard histograms:

- **1D conductance histogram:** Counts of conductance values (typically $\log_{10}(G/G_0)$) across all traces. The function calculate_his() in CaculateHistogram.py takes an array of conductance traces and returns bin edges and a histogram count array. It applies configurable binning and cut-offs for low/high conductance (to ignore unphysical out-of-range values). For example, by default it might exclude conductance below a certain logG (e.g., -6) or above a threshold. The result can be plotted as the aggregate conductance histogram.
- **2D conductance map (conductance vs. distance):** The function calculate_2d_his() produces a 2D histogram (heatmap) by accumulating all trace data points on a conductance-vs-length grid. This reveals how conductance evolves along the junction elongation. In the GUI, a 2D histogram is shown with distance (or time, if not actual distance) on the x-axis and conductance on the y-axis, color-coded by frequency of occurrence. The code (e.g., plt_2dIV() in CaculateHistogram.py) uses NumPy binning to create these heatmaps.

When histograms are generated, the software can also perform peak fitting. The module defines a Gaussian fitting routine: gaussian_fit(x, y) which normalizes the histogram y and uses non-linear least squares (scipy.optimize.curve_fit) to fit a Gaussian peak. This can be used to estimate the most probable conductance peak and its variance from the 1D histogram. A similar function gaussian_fit_2his exists for fitting multiple peaks or special histograms. Fitted peak parameters might be displayed or used for further analysis (e.g., comparing peak positions between clusters).

Clustering of Conductance Traces

Clustering helps automatically categorize traces (e.g., into different conductance pathways). InJunction offers multiple clustering approaches:

- **K-Means Clustering:** The simplest method, grouping traces by similarity (in features like principal components or histogram shape). The code imports scikit-learn's KMeans and PCA. Typically, traces are transformed (e.g., each trace's binned conductance histogram or maybe a lower-dimensional representation via PCA) before clustering. The clustering() function in CaculateHistogram.py likely runs KMeans with a specified number of clusters. If clusters_way=0 (an indicator in the code), KMeans could be the chosen method. The result is cluster labels for each trace.
- **Gaussian Mixture Models (GMM):** The code also imports

sklearn.mixture.GaussianMixture. If enabled (clusters_way=1 perhaps), the traces can be clustered by fitting a Gaussian mixture in the feature space. GMM can capture more complex cluster shapes than KMeans (which assumes spherical clusters).

- **Time-series clustering:** The presence of tslearn.clustering.TimeSeriesKMeans and KShape indicates support for treating each conductance trace as a time-series. *K-Shape* is a shape-based clustering algorithm for time series, aligning patterns regardless of phase. In practice, if clusters_way is set to use time-series clustering, the raw traces (conductance vs. distance arrays) are fed into these algorithms to cluster by the shape of the conductance decay profile. This is useful for identifying distinct plateau patterns that are not captured by simple histograms.

Selecting Number of Clusters (CH Scan): To assist in choosing an appropriate cluster count, the project implements a **"CH scan"** – using the Calinski–Harabasz index. This index (a ratio of between-cluster to within-cluster dispersion) is computed for different cluster counts and the optimal count is where CH is maximized. In code, sklearn.metrics.calinski_harabasz_score is used. The GUI's *CH Scan* button triggers a loop that clusters the data for K = 2,3,⋯ (within a range) and calculates the CH score for each. The result is shown as a plot of CH score vs. number of clusters, and the maximum suggests the best cluster number. This automates finding a reasonable cluster count for KMeans/GMM.

Executing Clustering & Results: When the user performs clustering (with a chosen method and cluster number), the software returns a label for each trace. The chose_cluster() function encapsulates running the clustering and returning the labels. For example, new_function.py calls labels = chose_cluster(conductance_normalized, clusters_way, n_clusters). After obtaining labels, the traces are split by cluster: the function devide_cluster() creates subsets of the conductance data belonging to each cluster. The GUI then can display separate histograms for each cluster (to show distinct peaks per cluster). Indeed, after clustering, InJunction plots the 1D histogram of each cluster's traces side by side for comparison (each cluster's histogram often highlights a different conductance peak, indicating different molecular configurations). These plots are arranged in the UI for easy visual assessment of clustering quality.

Merging and Saving Clusters

The **Merge saving** feature allows the user to manually merge clusters and save the merged dataset. After initial clustering, one might decide that two clusters represent variations of the same physical phenomenon and should be combined. In the UI, checkboxes or toggles next to cluster histograms let the user select which clusters to merge. When the user confirms, the selected clusters' trace data are concatenated. The code's divide_merge(judgment_list, ...) function is responsible for this. It takes a boolean list (judgment_list) indicating which batch (cluster) to include, and concatenates the corresponding conductance, distance, and length arrays into one dataset. The merged data is then treated as a single group for subsequent analysis or exporting.

The merged dataset (or any dataset at hand) can be saved to disk. **Saving Data:** The project uses NumPy's .npz format to store arrays. For example, after processing I–V or clustering, np.savez is called to save arrays like distance_array, conductance_array, and length_array (which may represent different things depending on context). Each saved .npz bundle can later be reloaded for analysis (the GUI offers a "Open file" dialog for .npz files). Saving clusters individually allows researchers to extract traces of interest for further analysis in external tools if needed.

Segmentation and Length Analysis (Subsection Cacu)

The **Subsection Analysis** page provides tools to analyze how conductance distributions change over time (or over trace index) and to examine plateau lengths at specific conductance values.

Batch-wise Conductance Distribution ("Time Histogram")

One feature is to split an ordered set of traces into sequential batches and observe distribution changes. For example, if the traces are in the order they were measured, this can diagnose stability over time. The function draw_divide(conductance, distance, length, ...) in new_function.py automates this batching. It divides the dataset into chunks of a fixed number of traces (e.g., 500 by default, controlled by cacu_num), computes a 1D histogram for each chunk, and then visualizes all chunks together:

- **"Time Histogram" image:** The leftmost panel (axes[0]) of the output is an image showing the stacked 1D histograms for each batch as a 2D color map. The y-axis corresponds to the batch index (chronological order) and the x-axis is conductance (log G). This image reveals if certain conductance peaks become more or less prevalent over time. For example, fading of a peak in later batches might indicate a change in experimental conditions or sample degradation.
- **Variance plot:** The next panel (axes[1]) shows the variance of each batch's histogram as a function of batch index. The code computes row_variances = np.var(hist_2d, axis=1) i.e. variance of each batch's histogram distribution. It then plots these variances vs. batch number. The average of these variances (var_variances) is also shown in the title. This plot helps quantify how much distributions differ between batches – a stable experiment would yield low variance for all batches (flat line), whereas large changes yield higher variance in certain segments.
- **Batch histograms:** The subsequent panels (axes[2] onward) show the actual 1D histogram bars for each batch individually. Each is labeled "batch 1", "batch 2", etc. This allows detailed inspection of each segment's conductance distribution. The bars are plotted with the same bin edges for direct comparison, and a grid is enabled for readability.

All these subplots are arranged side by side and saved as an image (draw_divide.png). The GUI likely displays this image or allows the user to scroll through the batch histograms.

The code supports **merging batches** as well. Suppose the user identifies that batches 1–3 have similar distributions and batch 4 is an outlier – they might choose to merge batches 1–3. This is achieved with the aforementioned divide_merge() function by passing a judgment list for batches. The merged result is a combined dataset (and could be saved or further analyzed as a single group).

Plateau Length Distribution Analysis

Understanding how long conductance plateaus last at certain conductance values is often crucial. InJunction includes an analysis for *plateau length vs. conductance*. The **"length subsection"** feature in the readme refers to this. The user can specify a conductance value of interest (a "peak" in log G, e.g., -3.5 corresponding to ~3×10^−4 G_0) and the software will measure plateau lengths around that conductance for each trace.

The function research_peak_length_his(distance, conductance, peak) in new_function.py computes an array of plateau lengths for a given peak conductance. For each conductance trace:

- It focuses on a narrow conductance window around the target peak (±0.1 in log G).
- It builds a small histogram (20 bins) of that trace's conductance values in this window, and finds the bin with the maximum count (i.e. the local most-populated conductance around the peak).
- It then finds all indices in the trace where the conductance is near that peak (within 0.1) and takes the largest index – effectively measuring how long the trace stayed at that conductance (distance index of the last occurrence of that conductance). Multiplying by distance increment (or using the index count as a proxy if distance is normalized) gives the plateau length for that trace's peak plateau.
- It appends this length to the stage list. After all traces, stage contains the plateau lengths distribution for that conductance peak across the dataset.

Using this, the peak_divide() and peak_draw_divide() functions create a histogram of plateau lengths for each batch of traces. This is analogous to the time histogram, but now the quantity being histogrammed is "length of plateau at X conductance" instead of conductance itself. The code divides traces into batches (like every 500 traces) and for each batch obtains the list of plateau lengths via research_peak_length_his. Then:

- It computes a histogram of those lengths for each batch (with configurable bins spanning a length range, e.g., 0–2 nm divided into 50 bins).
- peak_draw_divide then plots a heatmap of these length distributions vs. batch index (similar format to time histogram) and bar charts for each batch's length distribution. The x-axis is plateau length (nm) and y-axis is batch. This reveals if plateau lengths tend to get shorter or longer over time.
- The composite image is saved as draw_divide_peak.png and likely shown in the UI.

In summary, the Subsection Analysis module allows one to monitor experiment stability and plateaus: first by checking if conductance histograms drift over the measurement sequence, and second by analyzing how plateau lengths for a specific conductance peak vary. These tools are particularly useful for large datasets, helping identify systematic trends or drifts that might otherwise be missed.

I–V Data Processing Module

The **I–V Analysis** page and the backend module **ivDataProcessUtils.py** handle current-voltage curve data, which is another crucial part of molecular electronics experiments. Often, I–V curves are measured on single-molecule junctions (for example, after forming a plateau, a voltage sweep is applied). This module processes such data to make it suitable for further analysis (like plotting and noise calculation). The workflow includes reading raw files, segmenting the curve, converting units, and removing artifacts.

Reading TDMS Files

Raw I–V data are often recorded via LabVIEW or similar in National Instruments **TDMS** format. The function IVDataProcessUtils.loadTMDSFile(filePath) uses nptdms to read the file. A .tdms file can contain multiple channels; here it's assumed each file holds one I–V measurement with channels for bias voltage, current, and possibly measured conductance. For example, loadTMDSFile opens the TDMS and reads:

- **BiasVolt** = channel 0 (the applied voltage array over time)
- **Current** = channel 1 (recorded current corresponding to bias)
- **Cond** = channel 2 (conductance, possibly pre-calculated as I/V or measured via a transimpedance amplifier)

It returns these as NumPy arrays. The code includes comments in Chinese describing these steps (e.g., “读完数据就是 numpy 数组” meaning the data is read as a NumPy array).

If multiple TDMS files are selected in the GUI (e.g., an entire directory of I–V curves), the program will load each, process them, and perhaps combine results or save them separately. The UI shows the count of files loaded (printing “Number of TDMS files found: X”). Each file is then processed through the steps below.

Identifying Forward/Reverse Scans (Hysteresis Separation)

Many I–V measurements involve sweeping voltage from 0 up to some value and back to 0 (to check for hysteresis). The raw data is one continuous sequence of voltage and current. The method IVDataProcessUtils.hysteresis(filePath, bias_base=0.1, peakStart=-2.5, peakEnd=-5.5) splits this sequence into **forward** and **reverse** traces and cleans them. Key steps:

1. **Load data:** It calls loadTMDSFile internally to get biasVolt, current, cond arrays.
2. **Find voltage step regions:** It looks for the points where bias steps from 0.1 to 0.2 V and back. bias_base is 0.1 V here (half of 0.2 V). By finding indices where bias jumps by ±0.1 or ±0.2, it identifies the segment corresponding to a full sweep. Essentially, it detects the start index of the forward sweep (just after 0 -> 0.1 V step) and the end index of the reverse sweep (just before the final return to 0 V). This yields arrays biasVTrace, currentTrace, condTrace each containing the forward-and-reverse combined traces for all cycles found in the data.
3. **Ensure complete sweeps:** If any sweep is incomplete (didn’t return to 0 properly or missing segments), it is discarded. The code checks for at least two zero-bias points, etc., and drops any trace that doesn’t meet criteria.
4. **Determine true scan region:** Even within a complete sweep, initial or final flat sections at 0 V can be present. The algorithm finds the exact indices where the bias first leaves 0 and when it finally returns to 0 at the end. This is done by scanning through zero_idx (indices of zero bias) and identifying where the continuous run of zeros breaks (start) and where it resumes (end). It also checks the direction of the first sweep (positive or negative first) to adjust the end detection. The determined cutStart and cutEnd indices mark the actual sweep portion for each trace.
5. **Extract forward vs. reverse segments:** Using the cut indices, the code splits each sweep into separate forward and reverse scans. It pads traces to equal length temporarily to help identify segments, and uses differences in bias sign to categorize forward vs reverse segments. Essentially, whenever bias goes from increasing to decreasing or vice versa at a peak, that’s the turnaround between forward and reverse.
6. **Compute mean conductance at 0.2V:** The code calculates condCheckF and condCheckB – the mean conductance at the start (0.2 V in forward direction) and end (0.2 V in reverse direction) of the scan for each trace. This is used as a criterion: it requires these mean conductances to lie within [peakEnd, peakStart] (e.g., [-5.5, -2.5] in logG) to consider the trace valid. This filters out traces where the molecule wasn’t

present (very low conductance) or where an anomaly occurred.

7. **Finalize data:** After filtering, it compiles the final arrays: biasVData, currentData, condData for the valid scans, all truncated to the identified scan region (cutStart to cutEnd for each). It also computes meanCond for each full scan (average of condCheckF and condCheckB). Then it converts **current** data to log scale: currentData[i] = log10(|I|) + 6 (assuming I was in mA, this converts to nA and logs) and replaces $-\infty$ with -3 (an arbitrary floor). After these steps, hysteresis() returns currentData, condData, biasVData, currentData_so, meanCond. Here, currentData_so is a deepcopy of the original current (linear scale) for later use (e.g., generating I–V curves), while currentData is the log-scaled version.

At this point, we have clean forward-and-reverse separated I–V data for each measurement, in log scale for current and conductance.

Partitioning and De-capacitance Correction

The method IVDataProcessUtils.getPartitionData(currentData, condData, biasVData, currentData_source, meanCond, de_capicity=False) further splits the data into *forward* and *reverse* sets in a structured way. Its job is to take the full sweeps from hysteresis() and partition them by scan direction:

- It initializes lists for forward scans (...For) and reverse scans (...Reve).
- For each full scan (biasVData[i]): it finds the index of the peak bias (the maximum or minimum voltage) which separates forward vs reverse. It then splits the data at that peak into two segments. The code distinguishes whether the first segment is forward or reverse by checking if the first peak was positive or negative. It accumulates all forward segments across traces into the forward lists, and all reverse segments into the reverse lists.
- **De-capacitance:** If de_capicity=True, an extra step is applied to reverse scans. Capacitive current in I–V appears as an initial spike/decay especially in reverse (where the voltage is coming back to 0). The code attempts to shift the reverse current curve to remove this effect. Specifically, it finds the index of the minimum current in the reverse segment (min_index of currentData in that segment) and uses it to align the curve by a certain offset a. It then **shifts the current trace** by that offset (essentially moving the starting point inward so that the capacitive spike is centered). Simultaneously, it recalculates the conductance for that shifted segment: condData_i = (current_log - log10(|V|) + log_g0 - 1). Here log_g0 = log10(77.6e-6) (77.6 μS is 1 G_0) and the formula corresponds to computing $\log_{10}(G) = \log_{10}(|I|/|V|)$ in units of G_0 (the -1 seems to adjust for the scaling to G_0). This yields a corrected conductance array for the reverse scan that should be aligned with the forward scan conductance scale. If de_capicity=False, no shifting is done; reverse scans are taken as-is.
- After partitioning, the function returns a tuple of 10 elements: biasVDataFor, currentDataFor, condDataFor, biasVDataReve, currentDataReve, condDataReve, currentData_sourceFor, currentData_sourceReve, meanCond_sourceFor_new, meanCond_sourceReve. Essentially forward and reverse sets of bias, current (log and linear), conductance, and mean conductance.

Finally, IVDataProcessUtils.iv_process(tdms_file, bias_base, peakStart, peakEnd, de_capicity) ties it all together. It calls hysteresis() to get processed data for one TDMS file, then calls getPartitionData(..., de_capicity=⋯) to get structured forward/reverse data. It handles exceptions (printing an error if processing fails for a file) and returns the tuple from getPartitionData. In the GUI, this is invoked for each file. The result for each I–V curve can be stored or directly used for plotting:

- The forward scans can be visualized as conductance vs bias curves, and likewise for reverse scans. (The UI might plot them or just compute histograms.)
- Immediately after processing, the code computes histograms for the **I–V data**: The function cacu_3fig(V_data, logI_data, logG, I_data, ...) is used to generate three key figures from the processed I–V data. Internally, cacu_3fig (in ivDataProcessUtils.py) takes the arrays for voltage, log10(I), log10(G), and linear I, then:
    - It downsamples each trace to a fixed number of points (sample_point=1000 by default) for manageability.
    - It applies a smoothing to each I–V curve using LOWESS (locally weighted regression) from statsmodels to reduce noise.
    - From the smoothed I–V, it computes the derivative dI/dV and then takes log10|dI/dV| for each curve, collecting these arrays.
    - It then constructs:
        1. A 2D histogram of $\log_{10}$(I) vs V for the dataset (voltage on one axis, current on the other) using cacu.plt_2dIV. This shows the distribution of I–V points (essentially a heatmap of all curves).
        2. A 2D histogram of $\log_{10}$|dI/dV| vs V (likely via plt_2dIV_1div) to highlight where the conductance changes occur along the bias sweep.
        3. One or more 1D histograms such as the conductance histogram at specific voltages or the distribution of dI/dV values.
    - Gaussian fits might be applied to the resulting histograms to find peaks (for example, to find most common conductance at a particular bias).
    - cacu_3fig returns data such as hist_logI, hist_dI, their bin edges, etc., which main.py captures (in an object like self.his_For for forward data) and can plot or save.

The I–V page of the GUI likely allows the user to see the processed data results: it could display the overlay of all processed I–V curves, the histograms, and possibly the CH index if clustering I–V curves. It also provides options to **save the processed data**. Indeed, after processing, the code saves an .npz for each I–V dataset containing the arrays distance_array (here this would be the bias voltage array), conductance_array (the log conductance or current array), and length_array (re-purposed to store meanCond or some identifier). These .npz files can then be reloaded in the Cluster or Correlation pages as needed (the structure mimics that of break-junction data for compatibility).

In summary, the I–V module takes raw voltage and current data, isolates the usable portions of the sweep, converts them to log scale conductance, optionally corrects for capacitive currents, and produces cleaned datasets. It prepares visualizations like I–V heatmaps and histograms to help interpret the ensemble behavior of many I–V curves. This is crucial for identifying common I–V features (like gap opening, nonlinear transport regimes, etc.) across multiple junction measurements.

Fowler–Nordheim Transform (Single-Trace Analysis)

Fowler–Nordheim (FN) tunneling analysis is a specialized treatment of I–V data to extract barrier properties. InJunction lists **"FN transform"** as a feature, which likely refers to providing a tool to transform I–V curves for FN plotting. In an FN plot, one typically plots $\ln(I/V^2)$ vs $1/V$ for a tunneling junction; a straight line indicates Fowler–Nordheim tunneling, and its slope relates to the barrier height.

On the **Find Trace / FN Transform** page, the user might load or select a single I–V curve (possibly one of the processed ones) for detailed inspection. The software could then perform the FN transform and even fit a linear region. While the current codebase does not explicitly show a dedicated function for FN (the UI label was confusingly "Find trance"), the expectation is that:

- The user selects a trace (maybe by index or from a file).
- The software computes arrays for $\ln(I/V^2)$ and $1/V$ from that trace.
- It then fits a linear model to the region of interest (often the high-field region of the curve) to extract the slope.
- The GUI would plot the FN graph and possibly output the slope or the computed barrier height.

Even if this feature is not fully implemented in code, the interface is prepared for it. Developers or researchers can extend the new_function.py or create a new module to perform this analysis. Given that all I–V data can be accessed (the linear current data currentData_so and bias arrays are saved), implementing FN analysis would involve a bit of data manipulation and linear regression (which Python's SciPy or NumPy can handle easily).

In the context of the project, FN transform acts as a bridge between raw I–V data and physical interpretation (tunneling barrier). It complements the I–V processing by adding an analytical perspective. Since the project is modular, adding such analysis does not affect other components – one could write a fn_transform(trace) function and integrate it into the "Find Trace" page workflow.

Correlation Analysis of Conductance Fluctuations

The **Correlation Analysis** (not explicitly a separate page in UI, possibly part of Home or Cluster page) is handled by **correlation_analysis.py**. This module computes the correlation coefficients between conductance values at different points in the breaking traces. The concept is to see if, for example, having a high conductance at one region of the trace is correlated with conductance at another region – effectively a correlation of two points on the 1D histogram.

The correlation analysis expects as input a dataset of break-junction traces (distance vs conductance, or just the conductance traces after alignment to equal length). It likely requires the data in a NumPy array form (perhaps loaded from an .npz that contains conductance_array). The process is as follows in correlation_analysis.py:

- First, single_1d_his() (imported from CaculateHistogram) is used to project each trace onto a 1D conductance histogram vector. Essentially, for each trace (an array of conductance vs distance), it computes a binned histogram of conductance values. The result is a 2D array: each trace contributes one row (its histogram counts), and the columns represent conductance bins (e.g., from log G = -6 to 0 with 300 bins).
- Then cacu_delta(data_load, bins, low_cut_1Dcon, conductance_high_cut) is called to compute **δconductance histograms**. This function takes the 1D histograms

of all traces (conductance_array) and does: $\delta H_i = H_i - \bar{H}$, where H_i is the histogram of trace *i* and $\bar{H}$ is the mean histogram across all traces. In other words, it centers the data by subtracting the average distribution. This yields $\delta \text{conductance}$ matrix of the same shape (traces × bins) where each row now represents fluctuations about the mean.

- Next, cacu_cov(data, ...) computes the covariance (and eventually correlation) between histogram bins. It uses Einstein summation via NumPy's einsum to compute the covariance matrix: essentially, it does $\langle \delta H_i \delta H_j \rangle$ across traces. Concretely, it forms an outer product of each δH vector with itself and averages over all traces to get $H_{\mathrm{cov}}(j,k) = \frac{1}{N}\sum_{i} \delta H_{i,j}\delta H_{i,k}$. This yields a square matrix of size (bins × bins).
- It also computes the mean squared fluctuation for each bin: $\langle (\delta H_j)^2 \rangle$ (which is just the diagonal of the covariance matrix, but they compute it separately as mean_delta_conductance_square).
- It then converts covariance to **correlation coefficients**: $\mathrm{Corr}(j,k) = \frac{H_{\mathrm{cov}}(j,k)}{\sqrt{\langle (\delta H_j)^2 \rangle , \langle (\delta H_k)^2 \rangle}}$. This normalizes the matrix to values between -1 and 1. By construction, the diagonal will be 1 (each bin perfectly correlated with itself), and off-diagonals indicate how synchronous the fluctuations are between two conductance values.
- Before plotting, they flip the matrix upside-down (result = result[::-1]), likely so that high conductance bins appear at the top of the image (since array index 0 corresponds to lowest conductance bin, which is typically plotted at bottom). They define an extent for plotting axes in real conductance units (low_cut and high_cut values).
- Finally, they use plt.imshow to plot the correlation matrix as a color map (with a diverging colormap like *coolwarm*, centered around 0 correlation). They set the color scale limits (vmin, vmax default to -0.1 to 0.1 in code, can be adjusted). A colorbar is added and title "Correlation coefficient analysis" set. The figure is saved as an image file (cacu_cov.eps and cacu_cov.png) in a "png_images" folder. The function returns the matrix and extent (for further use, perhaps to overlay contours or to allow interactive use in a notebook).

From a scientific perspective, this correlation map tells us if, for example, fluctuations at low conductance are correlated with those at high conductance within the same trace. A prominent off-diagonal red patch might mean that traces which have above-average population in one conductance region also have above-average in another – indicating a possible link (maybe a two-step conductance drop happening together). Uncorrelated (blue/white ~0) means the occurrences are independent.

The correlation analysis would typically be run after a large dataset is loaded. The user might click a "Correlation" button (perhaps on the Home page or as part of cluster analysis) to perform this. The output image is then shown. This feature gives a more nuanced picture than the histograms alone, highlighting relationships between different parts of the conductance distribution.

Noise Power Spectral Density (PSD) Analysis

The **PSD Analysis** page focuses on *current noise* in the junction at various conductance states. Often, by analyzing the noise one can glean information about junction dynamics (e.g., 1/f noise indicates certain processes). The module for this is implemented in new_function.py via the getIntegPSD() function and related code.

Calculating PSD

The input to PSD analysis is typically a single long time-series of conductance or current at a fixed bias (or a segment of a trace during a plateau). For example, during a plateau one might record current as a function of time to analyze fluctuations. The user would provide this data – possibly by selecting a trace and a segment (the UI has fields for window size and cut index, etc.). Key parameters in getIntegPSD:

- **windowSize:** Number of data points to consider for PSD (the GUI likely provides this as "Window Size"). The code takes the first windowSize non-zero points of the input array GArray.
- **CUT_num and CUT_time:** These allow dividing the window into sub-intervals. Essentially, if CUT_num is set (e.g., 0, 1, 2…), the window can be further split. a = windowSize // (CUT_num + 1), then GArray = GArray_0[CUT_time * a : CUT_time * a + a] picks one segment out of the window. This way, one can compute PSD for different time slices of the data (to see non-stationarity). If not needed, CUT_num=0 and it just uses the whole window.
- **Frequency range (freqLow, freqHigh):** The user can specify a frequency band over which to integrate the PSD. Commonly one might integrate 1–1000 Hz region to quantify low-frequency noise. These are passed in and used later to integrate.

The function then computes the mean conductance of the selected segment (gMean = np.mean(GArray)) for reference.

For the PSD itself, two modes are available:

- Using **Matplotlib's mlab.psd** function, or
- Manual FFT computation.

If mode="mlab", it calls mlab.psd(GArray, NFFT=N, Fs=sampFreq, ...), where N is the length of the segment and sampFreq is the sampling frequency (provided via GUI, e.g., "Frequency = ... Hz"). This returns psd and freqs arrays. Notably, scale_by_freq=False is set (so that the PSD is not density-normalized by frequency, since they integrate manually) and pad_to=N to avoid zero-padding. The code then multiplies psd by 2, likely to match the convention of the manual method (because mlab.psd for one-sided PSD might output half the power).

If mode="fft" (or anything not "mlab"), it performs manual calculation:

- Computes the FFT: fftRes = fft(GArray) (using SciPy or NumPy FFT).
- Takes absolute value and normalizes: resNorm = |FFT| / (N/2) and then halves the DC component resNorm[0]. The result resNorm is effectively the one-sided amplitude spectrum.
- Constructs frequency axis freqs = linspace(0, Fs/2, N/2) and takes the first N/2 points of resNorm as oneSideFFT.
- Computes power spectral density as psd = oneSideFFT ** 2 (since power is amplitude squared).

Now psd and freqs represent the one-sided power spectral density of the signal.

Integrating PSD and Output

The code then finds the indices of freqs that lie between freqLow and freqHigh. It integrates the PSD over that band using the trapezoidal rule (np.trapz) to get a single number: **integPSD**. This number is essentially the total noise power in the specified frequency range for that time segment. It often corresponds to "low-frequency noise amplitude" when freqLow is near 0.
The function returns a tuple (integPSD, gMean) – the integrated noise and the mean conductance of that segment. If anything fails (exception), it returns (0.0, 0.0) to indicate an issue.
In the GUI, the PSD page allows setting parameters (window size, frequency band, number of segments). The typical use-case:

- Load a dataset or trace.
- Specify sampling frequency (so the x-axis of PSD is correct).
- Specify window size = how many points of the trace to analyze (ensuring stationarity).
- If interested in how noise evolves, specify CUT_num to split that window into multiple parts (and perhaps run the calculation for each part sequentially).
- Set the frequency band of interest (e.g., 1–1000 Hz).

The software likely then calls getIntegPSD for each segment (from CUT_time=0 up to CUT_num) and collects results. It builds lists of list_gMean and list_integPSD (these were seen referenced in main.py). Each segment gives one (gMean, noise) point. These can be plotted as a scatter: mean conductance vs. noise power. For instance, one might see that at higher conductance, noise power is larger (due to more atoms in contact or etc.), or any correlation. The program plots such a relationship or at least saves it. In main.py, after computing these lists, they save images integPSD_png_path and maybe single_cut_png_path or list_minN_png_path which could be related plots. For example:

- A plot of **noise power over time segments** (if one just wants to see noise vs segment index, maybe single_cut_png).
- A plot of **noise power vs gMean** (scatter plot, possibly labeled as "minN" or such, not entirely clear). It might plot all points and perhaps also highlight the minimum noise point (minN).

Given the context, likely the PSD page displays:

- A graph of PSD (maybe the actual PSD curve for one segment).
- A summary of integrated PSD vs conductance: e.g., a chart showing how noise varies with conductance level.

From the code structure, it looks like they leaned towards summarizing via integrated values rather than showing the full PSD curve to the user (which can be large). This is reasonable, as the integrated noise (like variance) is a single metric easier to compare.
In addition, if the user selects "mlab" vs "fft" mode via a toggle, they can compare the two methods (they should be theoretically identical for large N). The code multiplies the mlab result by 2 to match the manual method's scaling, indicating attention to normalization differences.

Machine Learning-Based Noise Classification

A novel aspect of the project is the inclusion of a trained machine learning model (**noise_signal_model.keras**) intended to classify traces as "signal" or "noise". This addresses the common challenge that not every recorded trace contains a true molecular junction event – some are just tunneling noise or artifacts. Automating the filtering of such traces can save

time.
While the integration of this model in the code is currently commented out (likely still experimental), the approach is as follows:

- The Keras model is loaded using tf.keras.models.load_model('noise_signal_model.keras'). This implies the model is a TensorFlow/Keras model saved in the .keras format (HDF5 or SavedModel). We do not have the architecture details here, but presumably it's a neural network trained on labeled conductance traces.
- Before prediction, each trace's data must be vectorized to the input shape the model expects. The code prepares tran_X (transformed X) for the model:
    - It collects the conductance traces into a list conductance. If each trace already has length 1000 data points, it uses it directly; otherwise it downsamples each trace to ~1000 points by slicing (taking every αth point, where α = len(trace)/1000).
    - Then for each trace in conductance, it takes every 2nd point ([::2]), resulting in ~500 points per trace (the comment suggests they wanted 84 points, which may be a mistake in code or comment). Possibly they intended a larger downsampling.
    - Each resulting array (mole_array) is normalized between 0 and 1 by min-max scaling.
    - These arrays are stacked into tran_X as a NumPy array of shape (num_traces, num_features).
- The model then would be used: predictions = loaded_model.predict(tran_X) to obtain a classification for each trace. For example, this could output probabilities or binary labels indicating *signal-like* or *noise-like*.
- The code was structured to then separate the data: initializing lists conductance_signal, conductance_noise, etc. for distance and length as well. A loop would iterate over each prediction and append the trace to one of those lists. For instance, if predictions[i] indicates noise, trace i's conductance/distance/length arrays go to the "noise" lists, otherwise to "signal" lists.
- After classification, the GUI could present a summary: e.g., "X traces classified as signal, Y as noise". It might allow the user to review the noise traces separately or simply exclude them from further analysis (like clustering). Possibly, one could save the filtered dataset (only signal traces) to a new file.

This ML model essentially implements an automated trace quality control. It is trained presumably on features of the conductance vs distance traces. For instance, true molecular junction traces have a characteristic plateau and sudden drop, whereas pure noise traces might have a very short or no plateau, or just exponential decay. A neural network can learn these patterns and classify accordingly – this is mentioned in literature as well (machine learning for STM-BJ trace classification).
Even though the provided code has the prediction lines commented out, it shows the project's forward-looking design: the hooks are in place to integrate ML. By updating or uncommenting those lines and ensuring the model file is present, users can leverage it. They would just need to ensure the data formatting is correct and possibly retrain the model with

their own data for best results (the model file presumably came from the authors' training on some dataset).

For researchers and developers, this means the codebase can incorporate advanced analysis like deep learning without major restructuring. The separation into functions (tran_X preparation, then model usage) is clear. One could replace noise_signal_model.keras with a new model or add additional models (perhaps for classifying types of junction events, not just noise vs signal).

Integration and Workflow

Putting it all together, **InJunction** covers a complete workflow for STM-BJ data:

1. **Data Input:** The user loads data either in raw form (e.g., TDMS files for I–V, or potentially text files for breaking traces) or in preprocessed form (.npz files). The program provides file dialogs and feedback on successful loading.
2. **Preprocessing:** Raw I–V data are processed (segmented, cleaned) via *IV Analysis* module before any further analysis. Break-junction traces might need alignment or baseline corrections (which XMe code had, but in InJunction it appears the assumption is data are already in a suitable numpy format for clustering).
3. **Exploratory Analysis:** On the Home or initial pages, users can look at overall histograms and correlation. For example, after loading a big dataset of conductance traces, one could directly compute the overall 1D/2D histograms and the correlation matrix to understand general features.
4. **Clustering:** Using the Cluster page, users can categorize the traces. They might first do a CH scan to decide cluster count, then perform clustering. The resulting clusters are visualized with their own histograms, making it easy to identify, say, a high-conductance group vs. a low-conductance group. The user can then merge clusters or exclude one (e.g., maybe one cluster is just noise traces – those could be dropped).
5. **Machine Learning filter:** (If enabled) the user could run the ML noise classification either before clustering (to remove noise) or after (to identify if a cluster corresponds to noise). This step is optional but would streamline cleaning.
6. **Detailed Analysis:** With cleaned, clustered data, the Subsection page allows deeper investigation: Are there drifts during the experiment (time histogram)? Does one cluster appear only in the early traces and disappear later? – the batch analysis can show that. Are plateau lengths different between clusters or over time? – the peak length analysis addresses this.
7. **I–V and FN Analysis:** If I–V curves were recorded for the junctions, the I–V page processes them. The user can then view distributions of I–V behavior for each cluster (for example, if cluster1 and cluster2 correspond to different molecular configurations, their I–V characteristics might differ; the histograms from cacu_3fig would show that). The FN transform page then lets the user extract barrier information from representative I–V curves.
8. **Noise Analysis:** Lastly, the PSD page gives insight into junction stability at the microscopic level. By measuring noise as a function of conductance (or time), one can tell if one configuration is noisier (e.g., a less stable contact might show higher 1/f noise). The integrated PSD vs gMean plot can be compared between clusters or over time.

Throughout, the software saves results (plots as images, data as NPZ files) for documentation or further external analysis. The modular structure (separate scripts for each major function group) makes it extensible. For instance, a developer wanting to add a new clustering algorithm (say DBSCAN for outlier detection) could do so in CaculateHistogram.py and call it from chose_cluster or a new function. Similarly, additional noise analysis (like frequency-dependent noise exponent) could be added to the PSD module.

Technical Insights and Extensibility

- The use of **scikit-learn** and **tslearn** means advanced algorithms are readily available (PCA, spectral clustering, etc.). The project already uses PCA and CH index – one could also compute other cluster validity metrics or do dimensionality reduction on the correlation matrix, etc., with minimal effort.
- The GUI design separates logic from interface. The heavy computations are in plain Python functions (which are easier to test and modify). The UI just calls them and then displays outputs. This is good software practice and means one can script these analyses headlessly if needed (by calling the functions without the GUI).
- The reliance on **numpy** for data and **matplotlib** for plotting ensures that results are reproducible and customizable. If the default plots are not publication-ready, one can find the data (often returned by functions or saved in known files) and re-plot with desired styles.
- The ML integration hints at future directions – possibly using neural networks to detect features in real-time. A next step could be to integrate training capabilities or to classify traces on the fly as they are loaded.

Conclusion

InJunction provides a comprehensive platform for STM-BJ data processing, combining traditional analysis (histograms, clustering, I–V curves) with modern techniques (machine learning classification, correlation mapping). By organizing the code feature-by-feature and file-by-file, it is relatively straightforward for researchers to follow the processing pipeline or to extend the codebase. Each Python module has a clear purpose: from **data loading** (ivDataProcessUtils) to **analysis algorithms** (CaculateHistogram, new_function) to **visualization and UI integration** (main.py with the Qt interface).

Researchers aiming to understand the conductance behavior of molecular junctions can use this tool to quickly cluster and categorize their data, verify consistency over time, and inspect outliers or interesting subsets in detail. Developers can extend it by adding new analysis functions or improving existing ones (for example, implementing the FN transform fully, or refining the ML model). The modular design and use of standard data structures make such enhancements feasible.

By documenting the project in depth, we ensure that new users can not only run the software but also trust and interpret its outputs. Each step – from raw data to final plots – is traceable through the code. This transparency is crucial in scientific software. In summary, InJunction serves as a powerful and extensible **workflow manager for molecular electronics data**, streamlining analysis and enabling deeper insights into STM-BJ experiments.

References: The UI framework by Wanderson Pimenta and algorithms inspired by XMe_DataAnalysis have been incorporated, under MIT License. Users of this software should cite the original authors accordingly, as noted in the README. All plotting and analysis routines are implemented using open-source libraries as listed in *requirements.txt*. Future updates may integrate more real-time analysis and user guide improvements, continuing the project's aim of being a "molecular electronics killer app" for data processing.