IMPERIAL

A Hybrid MATLAB–Python Framework for Accelerating Solid-State Nanopore Signal Processing

Author

Brandon Huang
CID: 01929812

Supervised by

Prof. Joshua Edel
Dr Aleksandar Ivanov

Department of Chemistry
Imperial College London
April 2025
Word Count: 9910

A Hybrid MATLAB–Python Framework for Accelerating Solid-State Nanopore Signal Processing

Brandon Huang

ABSTRACT

Solid-state nanopore sensing generates large, high-resolution ionic current datasets that challenge the efficiency of conventional MATLAB-based signal processing workflows. To address this, a hybrid MATLAB–Python computational framework was developed, optimizing five critical nanopore signal processing routines—resampling, moving-mean baseline correction, Whittaker smoothing, Poisson-based thresholding, and peak detection—by porting them from MATLAB into Python. After refactoring and optimizations, Python implementations utilized vectorized operations and just-in-time compilation to significantly improve computational performance. Integration between MATLAB and Python was facilitated through a custom shared-memory interface, with automatic handling of data structures and dimensional consistency. Although integration introduced overhead that reduced the performance gains for smaller-scale routines, substantial runtime improvements were observed for computationally intensive processes. Future improvements could expand performance analysis to include memory usage and more rigorous fidelity testing. The resulting framework demonstrates a practical and scalable approach to accelerating nanopore data analysis without disrupting existing experimental workflows.

1. Introduction

Early detection of disease biomarkers in complex biological fluids can be the difference between successful treatment and disease progression in cancer, genetic disorders, and infectious diseases. [1–5]. Conventional diagnostic assays often fall short due to high costs, limited throughput, and reliance on centralized infrastructure; those that do meet these logistical constraints frequently do so at the expense of the analytical performance required for early-stage screening [6]. To meet these demands, alternative sensing platforms have emerged: micro- and nanoscale sensing technologies offer compact, high-throughput alternatives that are also well suited for point-of-care applications [7, 8]. As diagnostic technologies continue to advance, computational tools supporting these analyses are similarly evolving to meet the needs of resource-constrained and distributed clinical environments [9, 10].

In recent years, the field of biosensing has expanded in response to growing demands for rapid, accurate, and cost-effective disease diagnostics [11, 12]. Within this landscape, single-molecule detection technologies offer new paradigms for early-stage ultra-sensitive screening by providing label-free analysis of target biomolecules [13–15]. Among these, nanopore sensors have emerged as a powerful tool for molecular detection by transducing molecular interactions into electrical signals and analyzing signal properties to elucidate analyte characteristics [16]. However, the high-throughput and transient nature of the resulting data introduces significant computational challenges, which can hinder their integration into diagnostic platforms [17].

Nanopore sensor technology is based on the principle of a Coulter counter, where particles suspended in an electrolyte induce transient impedance changes as they traverse across a nanometer-scale pore in the material [18]. This *translocation* generates a characteristic blockade signal that encodes molecular features such as size, shape, or conformation [19]. Nanopore technology can be divided into biological and solid-state types. Biological nanopores, which represent the earliest class of nanopore sensors, are formed by inserting pore-forming proteins into lipid bilayers, creating channels with fixed dimensions. These biological systems, while highly selective, are often limited by their fragile membranes, short operational lifetimes, and single-use formats [20].

In contrast, solid-state nanopores are fabricated in ultrathin membranes (typically of silicon nitride or graphene) and allow individual molecules to translocate through the pore under an applied electric field [21]. This offers enhanced mechanical durability, chemical robustness, and greater customization, enabling prolonged operation, chemical functionalization, and integration into diverse sensing platforms [19, 22, 23]. This experimental design, while customizable across platforms, generally follows a shared operational workflow, illustrated in Figure 1.

Extracting meaningful data from raw nanopore traces is a non-trivial task. The signals produced by solid-state nanopores are often noisy, brief, and highly variable, presenting major computational challenges. Thermal fluctuations, electronic artifacts, and baseline drift can obscure events of interest, complicating downstream analysis [24]. Additionally, variability in pore geometry and surface

Abbreviations: AOT: ahead-of-time; **ASLS**: asymmetric least-squares; **CPU**: central processing unit; **CSV**: comma-separated values; **FWHM**: full width at half maximum; **GPU**: graphical processing unit; **HDF5**: hierarchical data format version 5; **HPC**: high-performance computing; **I/O**: input and output; **JIT**: just-in-time; **MEX**: MATLAB executable; **PMF**: probability mass function; **RMSE**: root mean square error; **SHMEM**: shared memory module; **SNR**: signal-to-noise ratio.

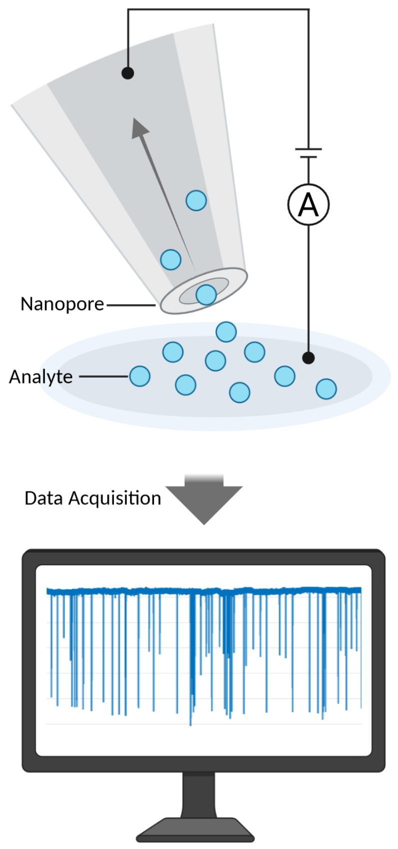

Figure 1: Nanopore Sensing Schematic. The core sensing principle common to a range of solid-state nanopore platforms, including the system studied in this work, are shown. Solid-state nanopore sensors (illustrated here as a nanopipette) detect individual molecules as analytes pass through a nanometer-scale pore, generating an ionic current trace. These brief blockade events within this trace encode molecular information.

charge can alter translocation dynamics and compromise signal consistency [21]. These challenges are amplified by the scale at which data is acquired: current traces are typically sampled at tens to hundreds of kilohertz frequencies, generating large, high-resolution datasets where individual translocation events may last only milliseconds or less. As such, extracting accurate, quantitative information requires advanced signal-processing techniques capable of handling both noise and volume efficiently.

Signal Processing. Signals arising from nanopore sensors are characterized by rapid, transient dips in the ionic current, each corresponding to the brief occlusion of a pore by an individual molecule. These events, occurring on microsecond-to-millisecond timescales, display considerable variability in both amplitude and duration—attributes that are highly sensitive to the physicochemical properties of the analyte and the surrounding ionic environment [25]. Thus, upon acquisition, raw nanopore current traces must undergo processing to extract characteristics that are quantitative (event duration, amplitude, and frequency), qualitative (peak shape, symmetry, and complexity), and aggregated (overall signal blockage, population-level distributions). Such extracted features are essential for downstream analyses, including molecular identification, concentration estimation, and event classification [19, 26].

In typical nanopore experiments, even with deliberate efforts to reduce interference, noise sources such as amplifier instabilities, capacitive coupling, and mechanical vibrations can still introduce artifacts that mimic true events, especially in high-bandwidth recordings [27, 28]. Because the events of interest are both brief and subtle, even minor distortions like baseline drifts or burst noise can critically undermine detection accuracy, necessitating processing strategies that preserve the event while suppressing noise artifacts [25, 29].

A robust signal-processing pipeline for nanopore data must address several core tasks that collectively ensure reliable detection and quantification of molecular events. A signal-processing pipeline for nanopore data typically encompasses four core steps [30]:

- **Data Cleaning:** to mitigate slow drifts and reduce systematic noise
- **Event Detection:** to isolate events caused by individual molecular translocation
- **Feature Extraction:** to detect or quantify signal parameters
- **Analyte Characterization:** to relate extracted signal features to the properties of the translocating analyte

Beyond these routines, the software executing the pipeline should also exhibit the following: scalability to process large volumes of data quickly and reliably; robustness against a wide range of artifacts; and reproducibility by delivering consistent results under variable conditions and computational environments. These requirements inform the integration of both classic methods (filtering, statistical thresholding) and more advanced algorithms (curve fitting, machine learning) to optimize the balance between computational efficiency and data accuracy [30].

To meet the analytical demands in signal processing, efficient software tools are essential. The core processing routines examined in this work were originally developed in MATLAB (2024a) as part of *The Nanopore App*, a platform used in selected solid-state nanopore sensing studies [31–33]. MATLAB remains widely used in academic signal processing due to its accessible syntax, extensive built-in libraries, and strong visualization capabilities [34]. Its design supports rapid prototyping, especially in early-stage method development, where iterative exploration often takes precedence over raw execution speed.

MATLAB is a *high-level programming language*, meaning it allows programmers to write abstract, human-readable code without managing a computer's system details. This differs from a *low-level programming language*, which provides finer control over hardware operations, but requires the programmer to navigate esoteric commands and manage granular tasks such as memory access, data types, and processor instructions. However, for further optimization, MATLAB achieves much of its performance through precompiled *low-level* implementations in C or Fortran, enabling efficient execution of numerical routines. Despite this, as nanopore datasets increase in size and complexity, limitations in MATLAB's data structures and memory management become more apparent [35]. The language's metadata-heavy architecture can lead to high memory usage and reduced runtime efficiency, particularly in workflows involving repeated operations across long current traces. While MATLAB offers built-in tools for performance tuning, it can struggle with large-scale or computationally intensive workloads that require sustained speed and scalability.

To address MATLAB's performance limitations in large-scale signal-processing workflows, this project explores the use of Python as an alternative implementation platform by reimplementing computationally demanding routines in this language. Although Python is also high-level, its open-source nature and modular design have led to its increasing adoption in scientific computing. Python provides significant flexibility for optimization through its extensive ecosystem of numerical and scientific libraries—such as `NumPy` and `SciPy`—which accelerate repetitive tasks and enable scalable data analysis [36]. Its stable compatibility with multi-core processors and GPU acceleration further enhances its suitability for handling large nanopore datasets efficiently [37].

Unlike MATLAB, however, Python relies more heavily on its dynamic features, such as flexible data typing and and more object-oriented design. These features introduce additional verification and memory management steps, particularly during repeated or large-scale computations. This results in computational *overhead*, meaning additional time or memory consumption beyond the core operation. Thus, achieving performance comparable to MATLAB often requires deliberate optimization of Python code or the use of tools that bypass these bottlenecks. Nonetheless, its modular structure and support for collaborative development make it well-suited for building maintainable and extensible workflows [38]. In the context of nanopore signal processing, these features can enable higher throughput, more responsive analysis pipelines, and improved adaptability to evolving experimental requirements.

Additionally, recent developments in solid-state nanopore research reflect a growing trend toward the adoption of Python as a platform for signal processing. Its versatility and extensive ecosystem support a range of applications. Examples include dynamic threshold correction and event detection [39], analyte classification using deep learning [40] and machine learning [41], real-time analysis pipelines [42], and synthetic signal generation for model training [43]. Collectively, these developments illustrate a broader shift driven by increasing data volumes and the complexity of modern experiments. Thus, exploring a migration to Python within established MATLAB-based platforms may offer benefits in long-term sustainability.

Aims and Objectives. In light of these challenges and opportunities, this work sets out to improve the computational efficiency of the existing *The Nanopore App*, with a particular focus on reducing runtime of specific signal processing routines. This will be achieved by porting core routines from MATLAB to Python and evaluating the impact of these changes on performance speed. The specific objectives are to: (1) identify bottlenecks in the original MATLAB implementation, (2) reimplement selected signal-processing routines in Python, (3) benchmark execution speed and output consistency across platforms, (4) develop a coherent method to integrate Python into the MATLAB platform, and (5) assess the suitability of the new Python-based framework for future high-throughput applications. In doing so, this work aims to deliver an efficient toolset for solid-state nanopore signal processing.

Given the tool-development nature of this work, an overview of signal processing is first described in detail to provide further context for the routines being optimized. Results are then presented in two parts: one focusing on performance optimization, and the other on cross-platform integration between MATLAB and Python; the discussion draws on both aspects to reflect on broader implications, challenges, and potential extensions of the work.

2. Signal Processing Overview

The core signal-processing routines used in *The Nanopore App* form the baseline for performance benchmarking in this work. This section presents the *final working versions* of signal-processing routines optimized in this work, acting as a reference to the computational routines that were optimized. The implementation and benchmarking of these routines in MATLAB and Python, respectively, will be evaluated and discussed in Section 3.1 (Computational Optimization), and the full methodology presented in Section 5 (Methodology).

For clarity, the term "routine" in this work refers to a distinct, self-contained step within signal processing, whereas "workflow" and "function" are used to describe broader or more granular levels of program or code structure, respectively. Each routine is described with a *platform-agnostic* logic, while recognizing their roles as programmatic functions.

It should be noted that the routines described reflect the design decisions of *The Nanopore App*, the interface examined in this project. As such, the focus of this work is not on proposing new signal processing strategies, but on porting routines onto a different programming platform for optimization. Descriptions will provide a brief rationale for the use of these specific techniques, but do not aim to comprehensively evaluate alternative approaches. The methods were implemented with minimal modification, with attention paid to preserving their behavior and output fidelity.

Additionally, the routines presented here do not constitute a complete pipeline for nanopore signal processing. Instead, they comprise the subset of procedures selected and developed within the project's timeframe. Hence, the routines discussed in this section define the practical scope of

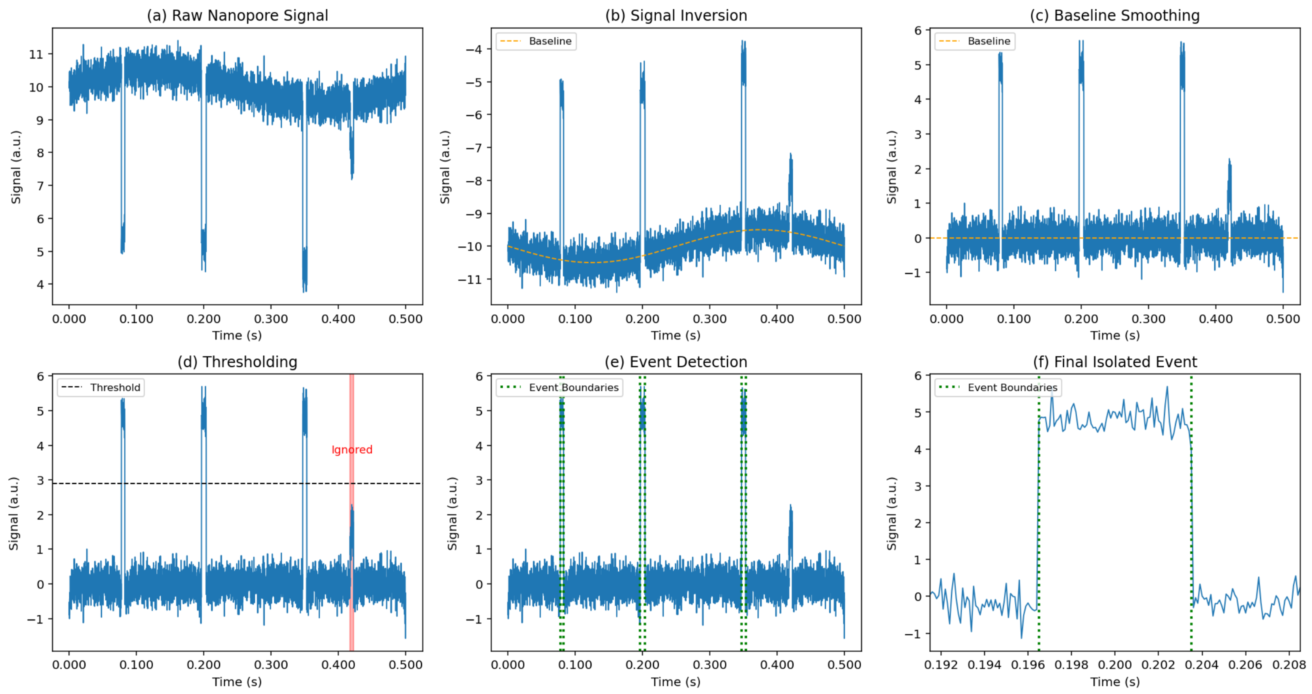

Figure 2: Signal Processing for Event Detection. A representative illustration of techniques used to extract translocation events from raw nanopore signals and prepare the data for further downstream analysis. The subplots show: (a) Raw signal with typical current blockades associated with single-molecule translocation in nanopore sensing. Where necessary, signals are first resampled. (b) The signal is inverted to convert current dips into peaks, ensuring a consistent signal direction suitable for peak detection algorithms. (c) A baseline smoothing filter is applied to correct for the baseline drift, centering the signal around a flat baseline. (d) A Poisson-based threshold is applied to exclude low-amplitude fluctuations, ignoring sub-threshold events. (e) Remaining peaks are identified as discrete events, with their start and end boundaries determined. (f) A single representative event is isolated for downstream feature extraction and analysis.

this work, which is summarized in Figure 2.

2.1. Preprocessing

Signal preprocessing functions serve to standardize, prepare, or transform raw data prior to analysis without altering its underlying information. These operations are essential for ensuring consistency across datasets and enabling robust, generalizable analysis in signal processing.

Signal Inversion. Prior to further processing, the raw current signal is inverted, transforming downward blockade spikes into upward peaks. This simplifies further event detection routines by aligning the signal with standard peak-detection algorithms, which are generally optimized for identifying positive-going deviations.

Resampling. Inconsistent or irregular sampling intervals can arise in raw nanopore recordings due to acquisition system limitations [44]. To ensure compatibility with processing routines that assume uniform spacing, a resampling step was introduced. The function first applies endpoint padding based on the mean of boundary values to reduce edge artifacts, then performs polyphase filtering using specified upsampling and downsampling factors. This preserves the timing and shape of transient features while enabling uniform analysis across recordings. Linear interpolation and other distortion-prone schemes were avoided to maintain the integrity of sharp transitions typical of burst events. The function takes (1) a raw signal, (2) an upsampling factor, (3) a downsampling factor, and (4) an optional padding value; it returns the resampled signal with the padding removed.

2.2. Smoothing and Baseline Correction

To prepare raw nanopore signals for reliable event detection, baseline drift and high-frequency noise must first be removed. Smoothing and baseline correction addresses these issues by stabilizing the signal and enhancing the contrast between transient burst events and background fluctuations.

Moving–Mean Baseline Correction. A symmetric moving average filter was implemented using a windowed cumulative sum approach, with adaptive windowing at the signal boundaries [45]. This approach is well-suited for large-scale signals, due to its simplicity and computational efficiency. Compared to more complex approaches such as higher-order polynomial smoothing, the moving average filter is preferred because it offers sufficient noise reduction while minimizing the risk of distorting short-duration burst events. The filter accepts three inputs: (1) the raw signal, and (2) a window size constant; it returns a baseline-corrected signal with negative values set to zero.

Whittaker Baseline Correction. Whittaker smoothing is an alternative approach to Moving Average Filtering, employed to correct for long-term baseline variations while retaining local signal features relevant to event detection [46, 47]. This method offers a balance between global trend suppression and local feature preservation, which is essential for resolving overlapping bursts in nanopore data. The smoothing was achieved using a penalized least squares method, which fits a smooth curve to the signal by minimizing both the difference from the original data and the roughness of the fitted curve. A smoothing parameter controls the trade-off: higher values suppress more noise but risk flattening real features, while lower values retain detail but allow more noise. Iteration controls were adjusted to ensure convergence of the solution. This approach provided flexibility across signals of varying event density and baseline characteristics [48]. The function accepts four inputs: (1) the raw signal, (2) a smoothing parameter (3) an asymmetry weighting factor and (4) the polynomial order of the difference penalty; it returns the estimated baseline.

2.3. Thresholding Methods

Thresholding is a critical first step in separating signal events from background noise. In the App's implementation, thresholding is performed in two stages: an initial histogram-based separation, followed by a probabilistic refinement using Poisson fitting.

Histogram Segmentation. An amplitude histogram of the baseline-corrected signal is first computed to define a preliminary boundary between background fluctuations and burst events. This method relies on a sufficiently high signal-to-noise ratio (SNR), where distinct amplitude distributions corresponding to noise and bursts can be separated [30]. Peaks in the histogram distribution are identified, and the positions of dominant peaks are used to establish an initial threshold for event detection.

Poisson Fitting. To further refine the separation, the histogram of signal amplitudes is fit to a Poisson distribution, which models the stochastic nature of translocation events occurring independently at a constant average rate [49]. Nonlinear least squares optimization is used to estimate the Poisson rate parameter, providing a statistical reference for distinguishing genuine burst events from random noise. Amplitude values falling below a specified probability threshold are retained as background, while rarer, higher deviations are classified as signal events. This probabilistic refinement adjusts and validates the preliminary threshold derived from histogram segmentation, providing a more robust foundation for subsequent event extraction.

Together, the overall thresholding function accepts four inputs: (1) the raw signal, (2) a standard deviation scaling factor, (3) a scaling offset, and (4) an override configuration dictionary that optionally replaces automatic threshold parameters.

2.4. Event Extraction

After suppressing noise and establishing threshold boundaries, peak detection serves as the critical step for quantifying discrete events within the nanopore signal. This phase is designed to reliably isolate significant bursts from the continuous signal, enabling subsequent downstream analysis to extract relevant quantitative metrics.

Peak Detection. Candidate translocation events are identified by segmenting the signal into contiguous time intervals where the amplitude exceeds a detection threshold. For each candidate event, the local maximum amplitude and corresponding time are determined, alongside the event's mean amplitude and integrated area under the curve. An initial filtering step removes events with invalid or zero duration. Optional refinement steps are applied to enhance detection accuracy: minimum and maximum event durations are enforced, and abnormally high-amplitude outliers are excluded. When enabled, a FWHM analysis is conducted, further refining event boundaries based on the half-maximum of each peak, and recalculating event width and area accordingly. Together, these steps ensure that only events matching the expected characteristics of true single-molecule translocations are retained, providing a robust basis for downstream quantitative analysis.

The peak detection routine accepts eight inputs: (1) the baseline-subtracted signal, (2) the estimated Poisson baseline, (3) a peak detection threshold, (4) the corresponding time vector, (5) the time resolution, (6) control parameters for full width at half maximum calculations, (7) event width and amplitude limiters, and (8) buffer correction settings. It returns eight outputs: (1) the timing of maximum burst peaks, (2) the peak amplitude of each event, (3) the mean amplitude across each event, (4) the start and end times of each event, (5) the integrated area under each event, (6) the full event segments, and (7) the indices of detected peak positions.

3. Results and Discussion

3.1. Computational Optimization

The computational performance of signal processing is a critical factor in the practical deployment of nanopore-based diagnostic systems. In this work, optimizations were performed at multiple levels, including refactoring, algorithmic profiling, and code-level acceleration techniques. Due to the hybrid MATLAB-Python development environment, particular emphasis was placed on validating output fidelity across platforms to ensure *functional equivalence*. This section presents a structured evaluation of computational performance, beginning with benchmarking methodology, followed by optimization and acceleration strategies of Python scripts using various techniques.

3.1.1. Benchmarking and Profiling

Before applying any optimization techniques, it was necessary to establish a robust framework for measuring runtime and verifying output consistency across MATLAB and Python implementations. This ensured that subsequent performance improvements could be meaningfully compared across both platforms. Full methodology for this data acquisition is presented in Section 5 (Methodology).

Runtime Evaluation Setup. To evaluate the performance of signal processing routines in both MATLAB and Python, duplicate benchmarking scripts were developed in each environment. This script repeats and times each routine, with the first run omitted to avoid overestimating runtime due to one-time initialization effects, and the remaining timings used to calculate the mean and standard deviation in execution time.

To test individual routines from the *The Nanopore App* in isolation, their typical input conditions and environment were recreated before measuring performance. This allowed for accurate measurement of runtime performance without interference from unrelated processing steps. To achieve this, each routine was preloaded with identical input data and parameters prior to timing. In MATLAB, a custom script set up the pre-function state by saving variables to a temporary MATLAB workspace, which was reloaded before each run to simulate a clean start. Similarly, in Python, a custom suite of functions define inputs and prepare the environment, ensuring test conditions remain consistent.

While the timing approach was similar across both platforms, the Python workflow also included line-by-line performance profiling module. This is a valuable tool that shows the execution time of each line, allowing for the identification of slower parts of the code, guiding later optimization efforts at this scale.

Equivalence Validation. Due to differences in internal implementations between MATLAB and Python (such as default data types, indexing conventions, and numerical functions) identical routines may produce subtly different outputs. These discrepancies can also arise when functions are executed across multiple platforms during conversion. For example, although Python's `pyabf` module was initially used to read raw trace data, inconsistencies were occasionally observed. As a result, MATLAB's functions were used to read the files, and the data were subsequently imported into Python. Because many signal processing routines are sensitive to small numerical changes, it was important to ensure that outputs were meaningfully equivalent across platforms.

To monitor for major discrepancies during cross-platform development, a simple output comparison framework was used. After execution, key outputs were saved to standardized `.mat` files and compared for basic shape consistency and numerical similarity. This comparison provided a qualitative check that outputs remained broadly comparable during the transition process, although exact numerical fidelity was not guaranteed. Unless otherwise noted, functions discussed in the following sections were visually inspected to be structurally consistent across platforms.

3.1.2. Programming Environment Behavior

To reduce the analysis runtime of nanopore data, core signal-processing routines originally implemented in MATLAB were reengineered and optimized in Python. The objective was to maintain functional equivalence while improving computational performance. Optimization efforts focused on five routines: (1) moving average filtering, (2) Poisson-based thresholding, (3) peak detection, (4) Whittaker smoothing, and (5) signal resampling.

Table 1: Relative Python Runtime (Histogram Function). Percentage change in execution time of Python implementations relative to MATLAB. Uncertainties are reported as standard deviations ($N = 100$). Data sizes are indicated for small datasets ($n = 10^2$) and large datasets ($n = 10^6$).

Function (data size)	Naive Runtime (% change)	Optimized Runtime (% change)
Histogram (s)	-62.2 ± 0.9	>999
Histogram (l)	62.0 ± 3.7	-18.1 ± 2.5

During this process, notable differences in efficiency emerged across the routines. These variations prompted a deeper investigation into the underlying causes, highlighting how each programming environment—MATLAB and Python—manages function execution, memory allocation, and data handling.

Initial Observations on Performance. Despite producing numerically equivalent results, MATLAB and Python differ fundamentally in their execution models, which has measurable implications for performance. Both are high-level programming languages, and thus abstraction improves accessibility and development speed but introduces computational overhead. Recalling that Python may require targeted optimization, performance was expected to lag behind MATLAB for unoptimized routines.

Initial attempts to reimplement MATLAB routines in Python using one-to-one, line-by-line translations led to degraded performance. These "naive" implementations often failed to leverage Python's numerical libraries effectively and exposed the cost of Python's interpretive overhead. During development, several functions exhibited substantial slowdowns in their unoptimized Python form, but achieved substantial speed improvements, after applying targeted optimizations, which will be discussed in detail in Section 3.1.3 (Routine Optimization). In several cases, runtimes fell below those of MATLAB, particularly for computationally heavy operations. This finding highlights the limitations of naive porting strategies and sets the stage for the deeper optimizations discussed in the next section.

More nuanced observation emerged when comparing performance across different data sizes. Certain optimizations introduce a fixed computational overhead that must be incurred before performance gains are realized. However, as the size of the input data increases, the relative impact of this constant overhead diminishes, allowing the optimization to deliver improvements in net performance. As a result, some "optimized" functions performed worse than their naive counterparts when applied to small datasets. Conversely, other routines scaled more efficiently with input size and outperformed MATLAB only when operating on larger arrays, as shown in Table 1. Unless otherwise noted in subsequent tables, positive values (shown in red) indicate slower Python performance, while negative values (shown in green) indicate speedup.

This discrepancy highlights a critical threshold: while some optimizations improve performance in theory, their benefits may only emerge once the dataset is sufficiently large. Thus, not all optimizations are universally beneficial, and their effectiveness can be sensitive to the scale of the data. It is therefore essential to consider not just the theoretical efficiency of a given method, but also its context of use. This is particularly pertinent since in the signal processing routine, functions can act on the entire signal trace (data size up to 10^8), or just the data in the peaks itself (up to 10^4). Effective optimization requires empirical profiling and a careful awareness of when particular techniques are likely to deliver net performance gains, especially in workflows with variable or high-throughput data loads.

Data-Type and Indexing Discrepancies Inconsistencies between MATLAB and Python present challenges for maintaining numerical equivalence. *Indexing* refers to the way programming languages reference the position of elements within arrays or vectors. MATLAB uses 1-based indexing, meaning the first element of an array is accessed with index 1. In contrast, Python adopts 0-based indexing. This fundamental difference affects all position-based operations and requires explicit offset corrections in functions to ensure consistent alignment of data between platforms.

Additionally, numerical data types also required attention. MATLAB stores values as double-precision by default, while some Python libraries and data loaders silently default to lower-precision formats such as `float32`. To preserve numerical fidelity all data were explicitly cast to `float64` in Python.

Moreover, MATLAB distinguishes between row and column vectors (e.g., 1×3 vs. 3×1), whereas Python often treats vectors as one-dimensional arrays with no inherent orientation. This difference can lead to shape mismatches during translating operations, such as indexing, matrix assignment, or broadcasting, after passing variables between platforms. During optimization, array dimensions were carefully managed to ensure that ported logic maintained consistent behavior across both platforms.

In cases where structured output data were used during testing, more advanced solutions were required; and full integration details are discussed in Section 3.2.2 (Cross-Platform Data Exchange).

3.1.3. Routine Optimization

Improving the runtime performance of nanopore signal processing requires careful attention to how individual routines are implemented. This section focuses on *refactoring*—altering the code without changing its function—to reduce execution time while preserving output fidelity. By analyzing and restructuring key functions, the goal was to enhance efficiency without modifying the underlying logic of the original MATLAB implementation.

Some routines, such as Whittaker smoothing, already have well-maintained and highly optimized implementations available in the Python ecosystem—for example, the version provided by the `pybaselines` library [48]. In such cases, integration was straightforward: the optimized external function could be directly incorporated into the workflow with minimal modification, significantly simplifying development while preserving performance.

Vectorization and Loops. Many workflows in nanopore signal processing require repetitive operations over large current traces. In such cases, performance is strongly influenced by how efficiently repetitive tasks are handled. Many initial Python implementations mimicked MATLAB's control flow using explicit loops. While this approach maintains conceptual clarity, it is considerably slower due to Python's interpreted nature, where every operation within a loop is handled individually. The Python library NumPy, in contrast, uses *vectorization*, a technique in which operations are applied to entire arrays at once rather than per element. This approach also leverages optimized C backends to execute computations efficiently, significantly reducing overhead and improving performance.

To illustrate this, consider the task of injecting random noise into a signal array from a two-minute nanopore trace, with $n = 10^8$ items, denoted as `s`, in MATLAB:

```matlab
% MATLAB: ~0.57 s
s = s + rand(1, length(s))*1e-9;
```

While conceptually simple, some implementations in Python may perform the same task significantly worse:

```python
# Python (slow): ~10.8 s
s += [random.random() * 1e-9 for _ in s]
# Python (fast): ~0.64 s
s += np.random.rand(len(s)) * 1e-9
```

In Python, the first implementation involves a separate function call to `random.random` for every element, making it both memory- and time-intensive when scaled to large arrays. In MATLAB, the operation is internally vectorized, allowing MATLAB to execute efficiently by avoiding the overhead of per-element interpretation. And thus, to resolve this, the second Python implementation shows a fully vectorized approach using the NumPy module, which generates the entire array and applies the operation in-place. This version leverages NumPy's underlying C libraries to execute the operation with near-native performance, making it over 15 times faster and comparable to MATLAB's built-in implementation.

Other examples include replacing Python's base operations with their NumPy equivalents, resulting in significant runtime improvements due to NumPy's compiled backend:

```python
max_value = max(data)    # ~27.04 ms
max_value = np.max(data)  # ~0.22 ms (0.8%)

sum_value = sum(data)    # ~40.10 ms
sum_value = np.sum(data)  # ~0.38 ms (0.9%)
```

This principle of replacing interpreted loops and base functions with optimized array-wide operations forms the foundation of efficient numerical computing in Python. While NumPy greatly improves baseline performance, the choice of expression still plays a critical role in maximizing efficiency—particularly when working with large datasets. For other smaller-scale scripts, such rigorous performance tuning is often unnecessary, but at the scale typical of nanopore signal processing, these optimizations can yield substantial gains.

Array Aggregation and Manipulation. Signal processing workflows often involve filtering, reassigning, or compressing large numerical arrays—such as removing noise below a threshold or labeling events. In Python, these operations can be expressed in multiple forms of syntax, each with different implications for memory use and runtime.

For example, two common approaches applying a condition to array data are *boolean indexing* and *conditional assignment*, both can be used for filtering or mapping data, but they differ in how efficiently they handle memory and computation. Boolean indexing directly modifies only the elements that meet a condition, making it memory-efficient for in-place updates. In contrast, conditional assignment evaluates the condition across the array and creates a new output array by choosing between two value options for each element. Although conditional assignment constructs a new array, it is often faster as it is typically implemented using `np.where`, which leverages NumPy's optimized internal routines.

```
# Filtering (Boolean Indexing): ~7.51 ms
filtered = data[data > value]
# Filtering (Conditional Assignment): ~4.26 ms
filtered = np.where(data > value)

# Mapping (Boolean Indexing): ~1.20 ms
data[data < value] = 0
data[data >= value] = 1
# Filtering (Conditional Assignment): ~1.53 ms
data = np.where(data < value, 0, 1)
```

In practice, boolean indexing is often preferred when modifying an array in place and conserving memory. However, `np.where` offers a more concise and sometimes faster alternative when constructing new arrays or expressing conditional logic.

In nanopore signal processing, operations such as peak labeling, denoising, and event segmentation rely heavily on conditional array transformations. Understanding the trade-offs between syntax choices is important for maintaining performance, particularly when working with high-resolution traces over long recordings. In the developed functions, these optimizations were evaluated using a profiler to measure line-by-line execution times and inform the selection of the most efficient approach. Beyond such micro-optimizations, the same performance principles apply to larger algorithms, where efficient handling of data grouping, indexing, and conditional logic is critical.

Custom Function Implementations. In nanopore signal processing, event-level features—such as peak amplitudes, durations, and areas—are often extracted from raw signals on a per-event basis and aggregated. These aggregation steps involve grouping values according to the index of the event to which they belong; each data point is associated with a detected translocation event, and summary statistics are subsequently computed on a per-event basis. This procedure recurs frequently throughout the processing workflow. Reproducing this behavior efficiently in Python posed a significant challenge, particularly when attempting to match the functionality of MATLAB's `accumarray` function, which is central to the original implementation.

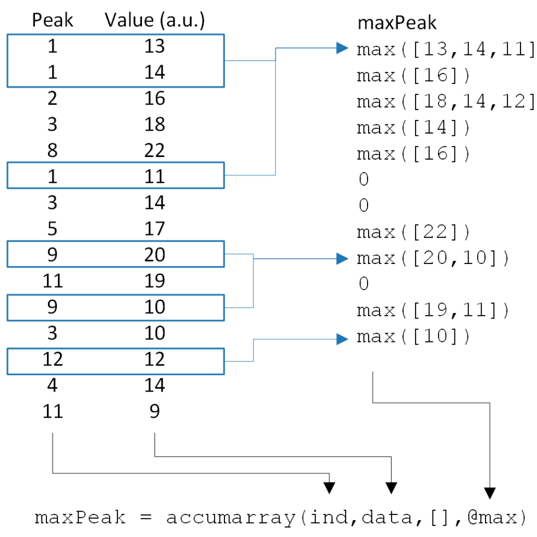

Figure 3: Accumarray in MATLAB. An illustrative example of the behavior of `accumarray` in a vector of processed data of 12 peaks. To find the maximum value reading for each peak, `accumarray` applies the max function to each group of values that have identical indices. Based on `accumarray` documentation [50].

In MATLAB, `accumarray` is a widely used function to facilitate grouped aggregation, allowing a user to input a function of choosing and apply it to elements grouped by corresponding indices (see Figure 3). This functionality is critical to *The Nanopore App's* signal processing where extracted features, such as durations, amplitudes, or areas, must be reduced to a single representative value per event. However, because MATLAB is a proprietary platform, the internal workings of `accumarray` are not publicly accessible. As a result, its algorithmic structure and edge case handling had to be independently reasoned and implemented. Although Python provides a rich ecosystem for numerical computing, it does not include a direct equivalent to `accumarray` in its core libraries.

Initial attempts at implementing the function involved simply recreating its logic: building groupings manually before applying aggregation. However, this quickly became unsuitable for large datasets. The reliance on dynamic Python objects introduced significant memory overhead, and the lack of vectorization led to performance degradation as input size increased.

To address these limitations, the algorithm was redesigned to treat different aggregation functions separately. This separation made it possible to optimize performance for common operations such as `sum`, `mean`, and `max`. These were implemented using NumPy's `np.bincount`, which efficiently performs grouped accumulation for integer-indexed

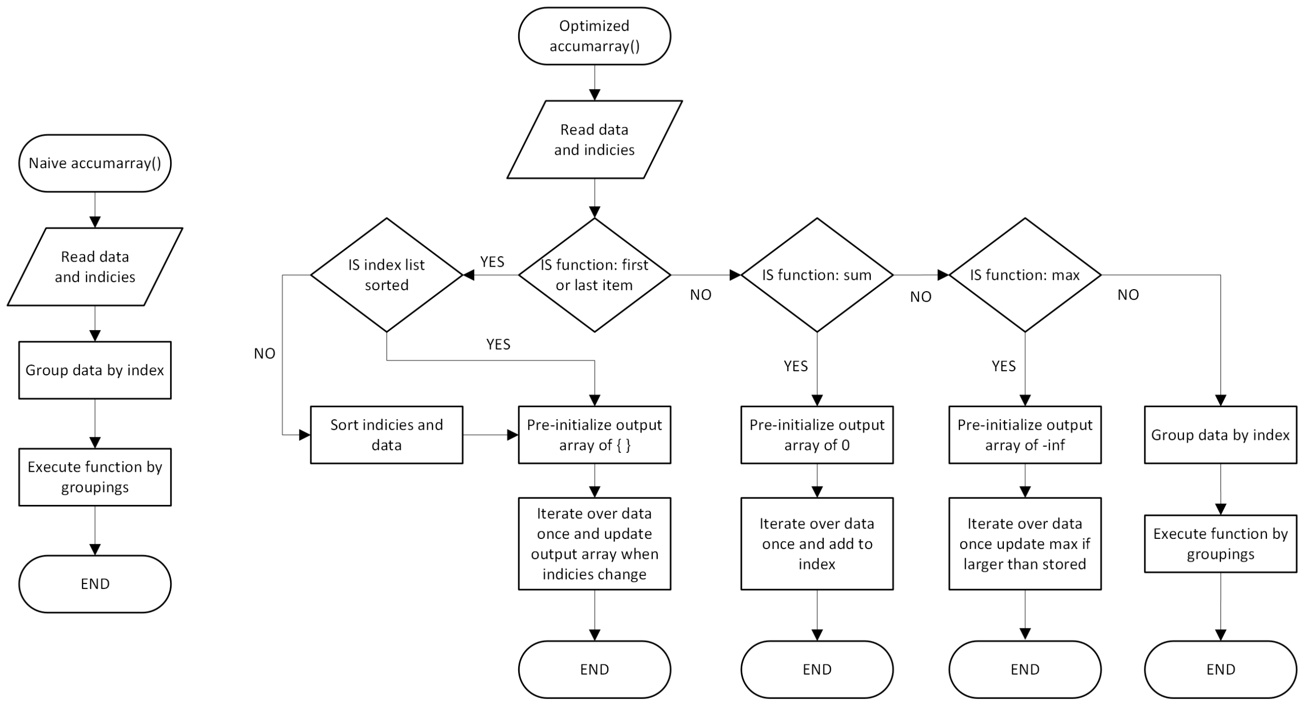
Naive accumarray()
Read data and indicies
Group data by index
Execute function by groupings
END
Optimized accumarray()
Read data and indicies
IS function: first or last item
IS index list sorted
YES
NO
Sort indicies and data
Pre-initialize output array of { }
Iterate over data once and update output array when indicies change
IS function: sum
Pre-initialize output array of 0
Iterate over data once and add to index
IS function: max
Pre-initialize output array of -inf
Iterate over data once update max if larger than stored
Group data by index
Execute function by groupings
END

Figure 4: Optimization Strategy for Accumarray Function. Flowchart comparing the naive and optimized implementations of the accumarray function in Python. The naive method groups data by index and executes the aggregation function across groupings. The optimized method selects specific strategies based on function type, significantly reducing computational overhead for large datasets.

data. While `np.bincount` mirrors the behavior of `accumarray` in these specific cases, it does not support the range of arbitrary or user-defined operations that `accumarray` allows. Additional logic was therefore introduced to support more general use cases.

For tasks that fall outside `np.bincount`'s capabilities, such as retrieving the first or last value in each group, the function uses preallocated arrays and single-pass linear scans. This strategy avoids dynamic structures while maintaining compatibility with large-scale datasets. In cases where a custom aggregation function is required, values are first collected into using Python's `defaultdict`, a special flexible container object, and the supplied function is then applied to each group. Although this fallback is slower, it enables full flexibility for specialized analyses.

By applying case-by-case optimization to different aggregation functions, the implementation achieves a practical balance between performance and generality; flowcharts of this restructure are shown in Figure 4. The most frequent operations are handled using fast, low-level routines, while more complex behaviors remain supported when needed. The benefits of these targeted optimizations are highlighted in a direct runtime comparison between the initial naive implementation and the optimized version. As shown in Table 2, the revised method can achieve more than a 99% reduction in execution time for large datasets, demonstrating the substantial impact of efficient aggregation strategies.

Table 2: Runtime comparison of `accumarray` in Python. Benchmarks were conducted on data of size $n = 10^6$. Timings are reported as mean ± standard deviation ($N = 100$ repetitions).

Method	Time (ms)	Δ Time (%)
Naive `accumarray`	1 512.90 ± 73.20	
Optimized `accumarray`	8.30 ± 0.40	-99.45 ± 4.82

3.1.4. *Compiled Acceleration Techniques*

To address the performance limitations inherent to interpreted Python code, compiled acceleration techniques were explored, particularly for repetitive, computationally intensive routines in signal processing. Unlike *compiled languages*, which are translated directly into machine code before execution, interpreted languages like Python and MATLAB execute instructions line-by-line during runtime. This introduces additional overhead from runtime checks, memory management, and dynamic typing. These features, while offering flexibility and ease of development, often result in slower execution, especially in numerical routines involving large datasets or frequent iteration. As such, strategies that mitigate these inefficiencies were explored by compiling selected routines into lower-level code.

Just-in-Time Compilation. One of the most effective strategies explored for performance improvement was just-in-time (JIT) compilation. JIT is a hybrid execution strategy that combines elements of both interpretation and ahead-of-time (AOT) compilation. In AOT compilation, code is fully translated into machine instructions before execution, typically at build time, enabling faster runtime performance

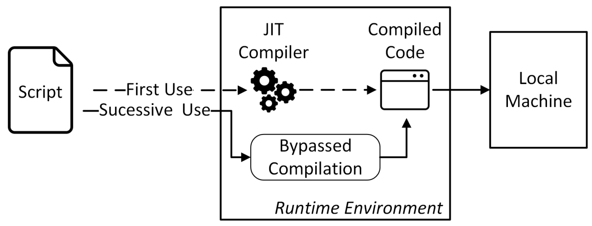

Figure 5: Just-In-Time Compilation Workflow. On first execution, the script is compiled into machine code by the JIT compiler. Subsequent executions reuse the compiled code, bypassing recompilation, thereby reducing runtime overhead during repeated operations.

but sacrificing flexibility. In contrast, JIT compilers dynamically identify performance-critical code during execution and compile it into optimized machine code at runtime. Subsequent uses of the function leverage the compiled code, resulting in significantly faster execution (see Figure 5). In Python, libraries like Numba enable JIT compilation with minimal changes to original syntax, making it particularly attractive for accelerating repetitive numerical computations while preserving Python's development flexibility.

JIT compilation using the `@numba.njit` function proved to be a practical and effective means of accelerating Python functions. By compiling numerical routines into machine code at runtime, Numba bypasses the interpreter overhead that typically slows Python execution.

However, the applicability of Numba was not universal. It only supports a limited subset of Python and NumPy features, and many high-level or dynamic operations are incompatible with JIT. As a result, most routines cannot directly adopt JIT compilation, or require significant refactoring by recreating Python functions to use simpler constructs that Numba can compile.

A representative example is shown below. The built-in `np.histogram` function offers convenience but performs slower on large datasets due to internal overhead. By contrast, a custom histogram routine compiled with `@njit` achieves substantially faster (five-fold) execution by using explicit loops and pre-allocated memory, but requires the logic to be reimplemented from first principles:

```python
# NumPy Built-In: ~50.5 ms
hist, _ = np.histogram(data, bins=bins)

# JIT-compiled: ~11.2 ms
@njit
def fast_histogram(data, bins, step):
    hist = np.zeros(len(bins))
    for value in data:
        bin_index = int(value / step)
        if 0 <= bin_index < len(hist):
            hist[bin_index] += 1
    return hist

hist = fast_histogram(data, bins, step)
```

This example demonstrates how selective JIT compilation can yield performance speedup, making it a highly effective optimization technique for selective numeric operations with predictable structure and large input sizes.

3.1.5. Alternative Acceleration Techniques

Although Numba offered a direct path to accelerating numerical routines, several other speedup approaches were also evaluated. Although these compiled acceleration techniques were not fully implemented into the final version, they were explored to assess their feasibility for future development. These investigations are summarized for completeness, highlighting technical barriers that informed their eventual deprioritization.

Parallelization Attempts. Parallel execution was also investigated using Numba's `@njit(parallel=True)` directive using `prange`, Python's `multiprocessing` module, and thread-based strategies. These methods offer potential speedups by distributing computations across CPU cores. However, for lightweight routines, such as histogram binning or event filtering, the overhead of task initialization and data transfer often exceeded the computational savings. Consequently, parallelization was found to be not suitable for these applications.

Cython and MEX Compilation. An alternative approach to achieving near-native performance was to compile Python routines into *binaries*—essentially machine-readable code—that can be accessed and executed from MATLAB. This strategy centered on embedding core functions into compiled modules using tools such as Cython. This module acts as an AOT compiler that translates Python-like code into optimized C code, which can then be compiled into shared libraries.

These compiled C modules were designed to interface with MATLAB through MATLAB executable (MEX) wrapping. In this context, *wrapping* refers to the process of enclosing a function within a small interface layer—called a wrapper—that handles inputs and outputs while leaving the function's internal logic unchanged. When applied to MEX files, wrapping allows compiled functions to be called from MATLAB as if they were native. MEX files enable this by allowing compiled C or C++ code to run within MATLAB, effectively bridging the two systems. This approach offers the dual benefit of acceleration and integration with the App's existing environment.

In practice, however, Cython-generated modules still rely on the Python runtime, and cannot be made fully standalone without significant engineering effort, such as embedding the interpreter and handling all cross-language memory and error resolutions manually. Attempts to implement this included generating shared libraries, writing export headers, and troubleshooting low-level interfaces using dependency analysis tools. Ultimately, the setup proved too fragile and time-consuming, and these methods were therefore deprioritized in favor of already developed solutions that required less infrastructure while still delivering meaningful performance improvements.

Although these alternatives could be promising for future extension, their practical challenges made JIT compilation via Numba the most effective approach for accelerating core functions in this project. Overall, these alternative strategies offer valuable insights into future optimization paths, but they played no role in the final performance results reported in this work.

3.1.6. *Performance Benchmarking Results*

The performance comparison focused on a series of core routines central to the signal processing in the *The Nanopore App*. Results for the routine optimization in Python are shown in Table 3. While some Python routines now outperform their MATLAB equivalents, some remain notably slower.

The benchmarking results highlight the varying success of optimization strategies applied to the Python implementations. In particular, routines that achieved comparable or superior performance to their MATLAB counterparts typically benefited from targeted code refinements. Extensive vectorization and direct array manipulations helped eliminate iterative bottlenecks and substantially improved runtime. Specific optimizations, such as minimizing unnecessary data copying, precomputing intermediate variables, and avoiding repeated function calls, also contributed to better performance.

The degree of performance improvement varied depending on the computational characteristics of each routine. Functions dominated by array-wide operations, minimal branching, and repetitive computation saw the greatest gains from vectorization and JIT compilation. In contrast, routines involving complex logic or short runtime durations benefited less, as either the optimization opportunities were limited or the compilation overhead offset any speedup.

Functions where Python remained slower generally reflected the advantage of MATLAB's highly optimized native functions. Built-in MATLAB routines such as `movmean()` and `lsqcurvefit()` leverage compiled, low-level optimizations that are difficult to replicate using standard Python libraries. Although custom Python equivalents were developed, such as a custom JIT accelerated moving average, these could not match MATLAB's intrinsic operations.

Overall, the results demonstrate that while aggressive code optimization can significantly narrow the performance gap, certain high-level operations in MATLAB retain a structural advantage due to specialized backend implementations that are challenging to match in Python without recourse to lower-level programming.

3.2. Cross-Platform Integration

As the signal-processing routines of *The Nanopore App* were originally developed in MATLAB, this work's decision to port selected components into Python introduced a major practical challenge: ensuring interoperability between two fundamentally different programming environments. Since MATLAB remains the primary development platform, this work required developing methods to integrate Python-based optimizations back into the MATLAB framework to enable their use and realize the associated performance benefits.

Table 3: Function Runtime Performance. Benchmarks were conducted using data representative of real-world signal traces. Timings are reported as mean ± standard deviation ($N = 100$).

Function	Platform	Time (ms)	Δ Time (%)
Moving Mean	MATLAB	89.2 ± 7.3	
	Python	190.7 ± 11.4	113 ± 13
Thresholding	MATLAB	164.7 ± 42.9	
	Python	74.1 ± 3.9	-55 ± 15
Peak Finder	MATLAB	417.3 ± 22.7	
	Python	307.7 ± 13.7	-26 ± 7
Whittaker	MATLAB	10 601.9 ± 155.5	
	Python	5 205.7 ± 46.9	-51 ± 2
Resampling	MATLAB	239.5 ± 54.9	
	Python	119.1 ± 2.7	-50 ± 11

A key practical aspect of the project must be addressed: establishing reliable and efficient ways for MATLAB and Python to exchange data and function calls. As nanopore signal-processing is iterative and data-intensive, poor integration could introduce significant latency or data inconsistency, undermining the performance benefits of the ported routines.

Several *programming* challenges inherent to cross-platform development have been previously discussed and resolved, including differences in indexing conventions, default data types, and structural representations of arrays. However, additional *architectural* challenges in the overall framework that supports cross-platform functionality still require alternative strategies, each with its own trade-offs in usability, portability, and runtime efficiency.

A key aim was also to explore whether these methods could support the eventual use of the Python routines in a standalone form within MATLAB, without needing ongoing support. To address these, multiple integration strategies were evaluated and implemented; each approach was designed with consideration for practical constraints such as ease of debugging, reproducibility, and compatibility with future extensions of *The Nanopore App*.

3.2.1. *Interfacing Python in MATLAB*

To integrate Python-based routines into any MATLAB app, several interface options were considered. Here, an *interface* refers to the mechanism by which programming environments exchange data and invoke functions. MATLAB provides native support for calling Python through the `py.` interface, which allows Python modules and functions to be accessed as if they were part of the MATLAB environment [51]. This approach is well-integrated for simple inputs but may become unstable when handling more complex workflows and would require additional supporting workflows to facilitate the cross-platform communication.

Another option explored was `pyrunfile()`, which executes an external Python script as a standalone process. While initially useful for prototyping Python code, it proved unreliable in practice: argument passing was inconsistent, and the script itself could not maintain internal variables independently, requiring additional external setup for every call. This made implementations fragile and difficult to maintain. A related alternative, `pyrun()`, enables execution of short Python code snippets directly from within MATLAB, but is mainly suited to one-liners or quick inline operations and lacks the structure needed for full routines.

Using `system()` calls to execute Python via the command line was also briefly evaluated. Although this approach is completely independent of the MATLAB–Python integration layer, it does not support variable exchange and offers no persistent Python state, making it unsuitable for most interactive applications.

Ultimately, the project adopted a class-based architecture using the first `py.` interface. By consolidating processing logic and data handling within a custom Python class, the integration remained modular and maintainable, while minimizing the complexity of calls made from within MATLAB. This structure also made it easier to add new functions to the custom ecosystem, and the modular design allowed individual routines in *The Nanopore App* to be tested in the future. While earlier concerns were valid, these were effectively addressed through the development of a self-contained Python module, details of which are discussed in the following section.

3.2.2. Cross-Platform Data Exchange

To support the hybrid MATLAB–Python workflow in *The Nanopore App*, a robust method for data exchange between the platforms was essential. Python-based processing workflows need to receive input data from MATLAB, execute signal-processing operations, and return results to MATLAB, ideally minimizing any additional processing. However, nanopore signal data is typically high in both resolution and volume, often involving millions of data points per trace. This placed significant demands on data transfer mechanisms, as even minor inefficiencies could lead to noticeable performance degradation when repeated across many processing steps. Identifying an approach that was both efficient and reliable became a central technical objective.

File-Based Transfers. Several common data exchange formats were initially explored to bridge MATLAB and Python. These included comma-separated values (CSV), MATLAB's proprietary data format (MAT), and hierarchical data format version 5 (HDF5).

CSV is widely used due to its simplicity and human-readability, and it is natively supported in both MATLAB and Python. However, it proved insufficient for this project. CSV files store data in simple two-dimensional tables and lack support for nested or complex data types such as structures or multi-dimensional arrays used during event detection routines. They also introduce significant overhead in the form of parsing and formatting, which becomes increasingly problematic when dealing with the large datasets typical of nanopore signal recordings.

MAT files, by contrast, are well suited to MATLAB workflows and support a broader range of data types. However, compatibility on the Python side requires additional modules such as `scipy.io` to read and interpret the files. While this solution is more flexible than CSV, it still involves disk input and output (I/O) and can lead to fidelity loss when handling non-numeric or structured variables. Furthermore, MAT files are closely tied to MATLAB's internal data structures, limiting their portability in cross-platform pipelines.

HDF5 was also evaluated, given its widespread use in scientific computing and its ability to store complex datasets with metadata. It allows for hierarchical structuring of data and supports large-scale numerical arrays. However, despite its efficiency for static storage, HDF5 is less well-suited for dynamic applications involving frequent read and write cycles [52]. In this project, many signal-processing routines required rapid, iterative access to small-to-medium-sized data blocks, which introduced noticeable latency when handled through HDF5. Moreover, managing file access, platform-specific compatibility, and version dependencies added additional complexity in a system designed to emphasize portability and modularity.

In summary, while each file-based format offered strengths, none aligned fully with the needs of this project. Many core routines required rapid, repeated data exchange between Python and MATLAB, and the overhead of explicitly creating and reading temporary files became a consistent bottleneck. These workflows often involved rapidly passing intermediate results between functions, where file-based transfers introduced unnecessary complexity and latency. As a result, a more lightweight and transient method of data exchange was explored.

Shared Memory. To address these limitations, the project evaluated, and ultimately adopted, a shared memory approach, enabling direct in-memory communication between MATLAB and Python. *Shared memory* refers to a reserved memory in the host system that multiple processes can access, allowing data to be exchanged without disk I/O or relying on intermediary files [53]. Compared to file-based formats, shared memory is particularly well suited for workflows with interdependent processing stages where data is frequently passed between the platforms.

This method is particularly advantageous in the context of nanopore signal processing, where workflows often involve iterative analysis. Each function call requires the exchange of large arrays, and when using file-based formats, the data conversion, disk access, and reloading steps collectively introduced measurable slowdowns. Shared memory eliminates this bottleneck by allowing both MATLAB and Python to access the same block of memory. In nanopore routines implemented in *The Nanopore App*, signal-processing functions are typically chained together with tight coupling between stages. For example, data may be smoothed, thresholded, and segmented in quick succession, with intermediate outputs needing to be passed from Python back to MATLAB's App environment. In this situation, shared memory ensures that high-frequency communication remains efficient.

A Custom Shared Memory Module. While Python offers built-in support for shared memory [53], MATLAB lacks native functionality for accessing or managing shared memory segments. To bridge this gap, a custom communication module was developed, building upon an available open-source `shared_memory` module, which backed by a library written in the Rust [54]. This Rust package integrates with other languages and provides a lightweight interface for allocating and accessing shared memory buffers, primarily intended for inter-process communication between platforms. It does this by passing the *pointer handle*—a reference to the location of data in memory—to allow MATLAB and Python to access the same block of memory directly.

However, the original implementation simply passes raw data without managing variable names, types, or dimensions. As a result, it provides no variable tracking, no coordination of multi-variable I/O, no protection against type mismatches, and no mechanisms for session control or automated cleanup. It also lacks the support to handle complex or nested structures commonly used in MATLAB. These limitations made it unsuitable for the modular, bidirectional workflows required in this project, where precise, structured, and language-agnostic data exchange was essential. And thus, a custom cross-language shared memory module (SHMEM) was developed to consolidate these transfers with the existing shared memory protocol, and its system architecture shown in Figure 6.

On the Python side, a dedicated `SHMEM_Processor` class was implemented to manage shared memory segments. This class handles all I/O exchange through a structured metadata protocol: each memory segment is accompanied by a JSON-encoded metadata record describing the variable's name, dimensions, data type, and timestamp. When the processing is complete, the updated data is written back to shared memory, and the metadata is updated accordingly. This ensures a fully decoupled interface between MATLAB and Python while maintaining the integrity and traceability of each data exchange.

On the MATLAB side, a suite of wrapper functions was written to mimic this protocol, handling the serialization of MATLAB variables into shared memory, invoking the appropriate Python functions, and reconstructing the results. *Serialization* refers to the process of converting data into a standardized format that can be stored or transmitted and later reconstructed. These wrapper functions also interact directly with the JSON metadata records to ensure consistency during data exchange. Special care was taken to support a wide range of MATLAB-native data types. This design directly addresses earlier concerns around complexity and fragility by encapsulating the communication logic and minimizing the burden on the MATLAB caller. To accommodate non-uniform data, such as nested cell arrays or structs used in event detection, the module uses JSON

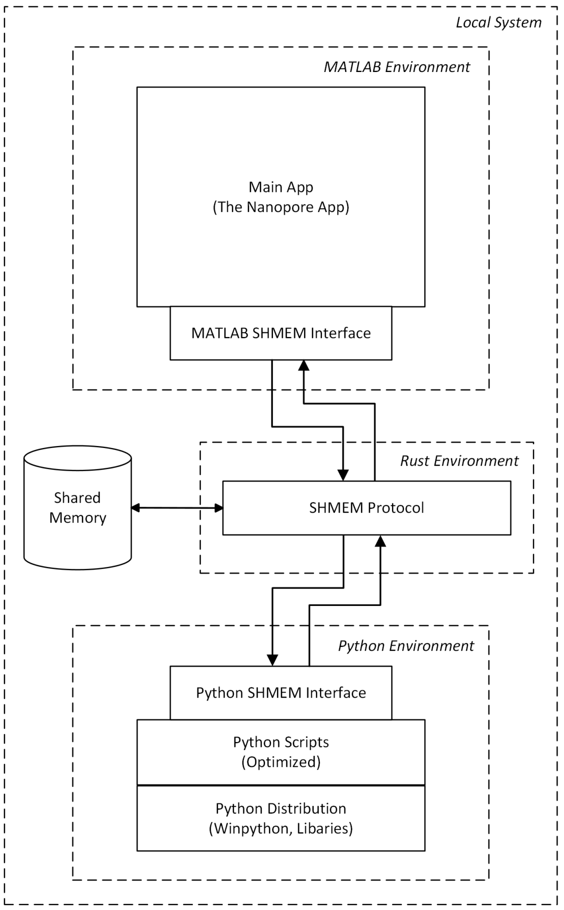

Figure 6: System Architecture of the Custom SHMEM Module. This shows the integration between MATLAB and Python via a shared-memory protocol. The MATLAB app passes data into a shared buffer managed by a Rust-based handler, which coordinates communication with a Python processing environment that includes its signal-processing scripts.

serialization with accompanying metadata that ensures accurate deserialization and type reconstruction in both environments.

For scalar values, strings, and other simple data types, instead of incurring the overhead of creating a dedicated shared memory segment for such variables, these values are stored directly within the metadata file using inline JSON encoding. This avoids the unnecessary overhead of full shared memory exchange for function calls involving small status values or scalar parameters, where shared memory offers no expected performance gain

Beyond the base protocol, a status function is invoked during app launch to verify that all components are functioning correctly. It also serves as a reference example for calling Python functionality from within the application. Additionally, a modular function registry system was implemented to support automatic loading of Python signal-processing routines, enabling MATLAB to detect and invoke newly added Python functions dynamically, as long as the corresponding `.py` scripts are placed in the specified directory. This allows future optimizations, or any other Python-based workflow, to be implemented directly into the *The Nanopore App*, ready for use.

This custom shared memory system was integrated into *The Nanopore App* to support high-throughput workflows with minimal modification to the existing application's logic. The final framework preserves MATLAB's interactivity while leveraging Python's performance for signal-processing routines, enabling a scalable foundation for future extensions of *The Nanopore App*.

3.2.3. Packaging and Portability

Beyond functionality, cross-platform development also raises practical questions around portability and reproducibility. *Packaging* refers to the process of organizing code, dependencies, and configuration into a structured format that can be easily distributed, installed, and executed on other systems. Ensuring that the Python environment could be deployed without disrupting existing setups or requiring complex configuration was a key consideration during the later stages of this work. Several strategies were explored to strike a balance between flexibility, isolation, and long-term maintainability.

Executable Creation. An early approach explored was the use of PyInstaller, a tool that packages Python scripts into standalone executables, or .exe files. These files include the Python interpreter and all necessary libraries, allowing them to run independently, without any installed Python environment. This was initially appealing as it would allow users to run Python-enhanced versions of *The Nanopore App* without installing Python or managing dependencies.

However, several drawbacks became apparent in practice. The resulting executables were large, often over 50 MB, due to bundling of large scientific libraries like SciPy, Numba, and additional bundled dependencies. More critically, startup and runtime performance suffered because of internal unpacking mechanisms and initialization overhead. Since App's pipeline involves frequent function calls and iterative data processing, this latency posed a real bottleneck. For these reasons, while PyInstaller provided a self-contained deployment route, it proved ill-suited for these performance-critical functions.

Portable Python Environment. To preserve performance while maintaining portability, a more effective approach involved using WinPython. This is a self-contained, Windows-only Python distribution that can run entirely from a local folder without installation or administrator privileges. Unlike PyInstaller, WinPython preserves the interpretability and flexibility of native Python scripts while avoiding conflicts with a user's existing system Python installations—or the lack thereof.

In this work, a custom WinPython environment was configured to include only the required libraries. It can be embedded directly into the deployment folder of *The Nanopore App* and accessed through MATLAB's `pyenv` function. The `pyenv` command allows MATLAB to interface directly with an external Python interpreter, enabling function calls between the two platforms. This setup ensured that Python scripts could be invoked as part of the MATLAB App, without any user configuration. Importantly, it also guaranteed that all users, regardless of system configuration, would be running the same Python environment, supporting reproducibility and easing future maintenance. In the context of tool development, this approach provided a balance between transparency, user control, and computational performance.

Constraints of Standalone Deployment. While multiple strategies were considered to encapsulate Python functionality within a single executable or library, practical constraints made full standalone deployment infeasible. Additionally, previous attempts to achieve full standalone functionality using Cython and MEX wrapping also proved impractical. These either introduced excessive overhead or failed to integrate cleanly with MATLAB's environment, especially when dealing with platform-dependent dependencies and dynamic typing.

Additionally, while WinPython avoids the startup delays associated with executables, the bundled WinPython environment still occupies approximately 300–400 MB, which is relatively large. However, this size overhead was considered acceptable given the elimination of runtime slowdowns and the improved reliability in the cross-system platform.

Moreover, MATLAB's architecture does not natively support deep Python embedding without relying on the Python interpreter itself. This makes full abstraction difficult to integrate and maintain. Given the modular and performance-sensitive design of *The Nanopore App*, the final deployment architecture intentionally avoids overcomplication. Instead, the reproducible setup using WinPython and structured shared memory, which offers reliability, maintainability, and MATLAB–Python interoperability.

3.3. Further Discussion

This project aimed to enhance the computational efficiency of solid-state nanopore signal processing workflows by porting critical routines from MATLAB to Python. It also addressed the modularity and maintainability of the system by integrating the two environments through a shared memory interface. While MATLAB has historically dominated experimental chemistry environments for its robust visualization and prototyping capabilities, it faces limitations when scaling to large, high-throughput datasets common in nanopore sensing. Python, with its extensive scientific libraries and increasingly optimized numerical backends, offered a promising pathway to extend and modernize computational capabilities.

Isolated *function* benchmarks, as seen in Table 3, has demonstrated substantial performance gains for optimized Python routines. Techniques such as vectorization, JIT compilation via Numba, and replacement of base operations with compiled NumPy functions enabled Python implementations to approach or even surpass MATLAB performance in several core tasks.

However, when tested within the full integrated *routine*—including shared memory data exchange and cross-platform system—performance dynamics shifted; the

Table 4: Fully Integrated Routine Runtime Performance. Benchmarks reflect the runtime of complete routines as executed through the integrated MATLAB–Python system, including shared memory exchange and cross-language invocation overhead. Timings are reported as mean ± standard deviation in seconds ($N = 100$).

Routine	Platform	Time (s)	Δ Time (%)
Moving Mean	MATLAB	0.089 ± 0.007	
	Python	0.498 ± 0.010	460 ± 37
Thresholding	MATLAB	0.165 ± 0.043	
	Python	0.430 ± 0.012	161 ± 42
Peak Finder	MATLAB	0.417 ± 0.023	
	Python	1.246 ± 0.035	199 ± 12
Whittaker	MATLAB	10.602 ± 0.156	
	Python	6.315 ± 0.141	-40 ± 1
Resampling	MATLAB	0.240 ± 0.055	
	Python	0.660 ± 0.064	175 ± 44

full benchmarks are shown in Table 4. Integration overhead, approximately on the order of hundreds of milliseconds per call, disproportionately affected short routines, such as moving average filtering and thresholding. In these cases, the fixed integration overhead increasingly outweighed the computational gains as the complexity and processing time of the function were smaller. Only computationally intensive routines, such as Whittaker smoothing where longer processing times offset the fixed overhead, maintained a significant performance advantage when fully integrated.

This finding highlights a key consideration for hybrid system design: optimization at the function level does not necessarily guarantee end-to-end workflow improvements, particularly when frequent cross-environment communication is involved.

Limitations. Several practical limitations were identified in the integrated solution. First, the system is currently limited to Windows platforms. Porting the project to macOS would require redevelopment of the SHMEM module and integration with a macOS-compatible Python distribution; however, both tasks are documented and expected to be relatively straightforward. Furthermore, while large-scale data processing benefits from Python integration, workflows involving predominantly short, fast operations may experience net slowdowns due to the integration overhead. This makes the system best suited for pipelines where computational routines dominate execution time, rather than highly interactive or real-time workflows.

While the project achieved its primary objectives, only five core routines were examined in depth. Although these represent major signal processing steps, they do not constitute a comprehensive evaluation of all possible nanopore workflows. As such, the generalizability of the optimization and integration strategies to other types of signal processing tasks remains to be fully explored.

Benchmarking efforts focused exclusively on execution time, without quantitatively assessing other critical performance metrics such as memory consumption, energy efficiency, or potential multi-threaded scalability. This narrowed focus was appropriate for the project's scope but limits the broader interpretation of system performance. Furthermore, all benchmarking was conducted on a single hardware platform and operating system, which may restrict the generalizability of the observed results across different computational environments. Finally, while functional equivalence between the MATLAB and Python implementations was verified, minimal downstream validation was performed to assess the potential impact of reimplementation on feature extraction accuracy, such as peak identification or event classification. Future work could address these gaps by expanding the range of routines tested, conducting more holistic performance profiling, and fully evaluating the scientific validity of the reprocessed signals.

Building on these observations, future work could aim to expand the system's cross-platform compatibility, enabling deployment across other operating systems. Additional optimization could involve compiling performance-critical routines using Cython or MEX interfaces to reduce cross-language overhead and improve suitability for real-time or latency-sensitive applications. Broader benchmarking, including memory usage, energy efficiency, and multi-threaded scalability, would provide a more holistic understanding of system performance.

Furthermore, extending the reimplementation framework to support a broader range of nanopore signal processing tasks, such as machine learning–based event classification and adaptive filtering, could enhance its relevance in evolving experimental workflows. Real-world validation against diverse experimental datasets would also be a critical step in establishing the robustness and scientific validity of the integrated system.

To ensure fidelity between MATLAB and Python outputs, more rigorous testing could be implemented. This may include systematic comparisons of extracted features, such as peak timing, amplitude, and event duration distributions, across real datasets. Statistical tests or distribution similarity metrics could be used to verify that the reimplementation preserves the underlying physical properties of the signal. Such analysis would complement execution-time benchmarking and increase confidence in the scientific accuracy of the optimized workflows.

Addressing these limitations defines clear directions for future work. Expanding cross-platform compatibility, exploring compiled interfaces to reduce overhead, broadening the range of optimized routines, and conducting real-world validation would all strengthen the utility and generalizability of the developed framework.

4. Conclusion

This project initially set out to accelerate signal processing routines within *The Nanopore App* by porting core functions from MATLAB to Python. Initial efforts focused on direct translation, but substantial performance gains were only achieved through structural optimizations such as vectorization, memory preallocation, and just-in-time compilation. These results highlight that effective cross-language optimization requires careful alignment between programming model, data structure, and runtime behavior. The integration framework developed here preserves existing GUIs and analytical pipelines while enabling backend enhancements, supporting hybrid toolchains without requiring major rewrites. The current implementation is limited to Windows and a subset of routines. However, its modular design allows for future expansion, cross-platform compatibility, and improved fidelity testing.

A standalone, hybrid MATLAB-Python framework was presented for signal processing in this work, which can be generalized to support integration of Python-based accelerations into existing MATLAB tools across scientific domains. While both MATLAB and Python rely on similar low-level optimizations to achieve high performance, Python offers broader advantages—including open-source extensibility, GPU support, and integration with modern computing tools—that make it particularly well-suited for data-intensive workflows. The integration framework presented here demonstrates how these strengths can be combined with existing MATLAB infrastructure to enable flexible, hybrid processing architectures aligned with the evolving demands of solid-state nanopore signal processing.

Acknowledgements

The author would like to thank Professor Joshua Edel and Dr. Aleksandar Ivanov for their invaluable guidance and support throughout the development and execution of this investigation. The author also wishes to thank Yilin Li for her introduction to the research group's core activities and experimental procedures, and Seshagiri Sakthimani for his support with programming and for providing valuable technical insights.

5. Methodology

All computational work was conducted using MATLAB R2024b and Python 3.11.9, with the primary libraries including NumPy (v2.1.0) [26], SciPy (v1.15.2) [55], Numba (v0.61.0) [56], pybaselines (v1.1.0) [48], pyabf (v2.3.8) [57], and line_profiler (v4.2.0) [58]. Timing measurements and performance evaluations were carried out on a computer equipped with an Intel i5-11600K CPU, 32 GB RAM, and running Windows 10. MATLAB operations were performed in a standalone desktop environment. Python scripts were executed in a Conda-managed virtual environment to ensure package consistency, or, when called from MATLAB, via a bundled WinPython distribution matching the same package versions. The full source code for all of the following methods is provided in the Supplementary Information.

5.1. Data Acquisition

Runtime performance was measured using custom timing scripts developed in MATLAB and Python. Graphical outputs were disabled during timing measurements to focus on computational performance.

In MATLAB, a `Timer.m` framework executed a pre-function to initialize the environment, a main function to perform the computation, and an optional post-function. A .mat workspace was saved and reloaded before each timed run to maintain consistent initial conditions. All timing was conducted with `tic` and `toc`, and performance was reported as the mean and standard deviation across repeated runs.

In Python, a corresponding `Timer_py.py` framework followed the same structure: a `pre_function`, `function`, and `post_function`. Timing was performed using the `time` module, with the first run excluded to account for setup overhead. Mean and standard deviation were computed over subsequent repetitions. Line-by-line profiling was optionally performed using the `line_profiler` module. For smaller-scale or inline timing tasks, the `timeit` module was used on synthetic data.

5.2. Signal Processing Functions

All signal-processing routines described below were implemented in Python; variable names were chosen to align as closely as possible with those defined in *The Nanopore App*.

Resampling (`resample.py`). Signal resampling was performed by first extending the baseline-corrected current trace `Count_2` with constant segments of length p, calculated as the mean of the first ten and last eleven samples. A polyphase filter (`scipy.signal.resample_poly`) was then applied using an upsampling factor `num` and downsampling factor `denom`. The added padding was subsequently removed to restore the original trace domain.

Moving–Mean Baseline Correction (`movmean.py`). Background estimation was performed by applying a centered moving–mean filter (Numba-JIT-accelerated) with window length k, offset by subtracting $0.75 \times \sigma$. The baseline-corrected trace was obtained by subtracting this estimate from the raw signal and clipping negative values to zero. Edge effects were handled by shrinking or growing the averaging window at the trace ends.

Whittaker Baseline Correction (`whittaker.py`). Baseline estimation was performed using the asymmetric least-squares (ASLS) algorithm from the `pybaselines` library, applied to the raw signal y. The smoothing process was controlled by the smoothing parameter λ, the asymmetry weight p, and the derivative order `order`. Up to 10 iterations were executed to ensure convergence of the estimated baseline.

Thresholding (`threshold.py`). The first difference of the baseline-corrected trace `Count_Base` was computed, with NaNs set to zero, and its mean absolute step, Δ, estimated (floored at 0.001). A bin vector $s_i = i\,\Delta$ spanning the amplitude range of `Count_Base` was generated, with Δ halved if fewer than 25 bins resulted. If the override flag in `ThrStep` was enabled, Δ and the maximum bin value were instead taken from user-specified parameters. A normalized histogram $H(s_i)$ was then computed using a Numba-JIT-accelerated routine, and its first bin was removed. A Poisson probability mass function, $\mathrm{Pois}(u;x)$, was fit to H using bound-constrained nonlinear least squares, yielding an estimated rate $\hat{u}$. The detection threshold was defined as $\text{thresh} = \left(\hat{u} + \text{STD}\sqrt{\hat{u}}\right) \times \Delta$, and the scaled Poisson rate as $\lambda_{\text{scaled}} = \hat{u} \times \Delta \times SO$. The function returns thresh, λ_{scaled}, $\{s_i\}$, H, and the fitted Poisson curve.

Event Extraction (`peakfinder.py`). The baseline-corrected trace `Count_Base` was first converted to NumPy arrays and cast to float64. Candidate bursts were identified by thresholding against the scaled Poisson rate, λ, and threshold, θ, with optional buffering (`Buff`) used to merge events separated by up to `Buff['Numb']` samples. The helper routine `PeaksBeta_V2` then labeled contiguous supra-threshold regions and, for each burst j, computed the peak time $T_{\max,j}$, peak amplitude $I_{\max,j}$, mean amplitude $\bar{I}_j$, start time $T_{\text{low},j}$, end time $T_{\text{high},j}$, area under the curve A_j, extracted segment trace X_j, and event index k_j. Post-processing filters specified by `WidthLimit` (minimum and maximum dwell times) and `CurrentLimit` (maximum $I_{\max}$ in nA) were applied to exclude spurious events. When full-width at half-maximum analysis was enabled (`FullWidthHM['On']`=1), half-maximum thresholds $\left(I_{\max}/\texttt{FullWidthHM['factor']}\right)$ were used to determine left and right FWHM boundaries, yielding refined durations and areas. The function returns arrays of $T_{\max}$, $I_{\max}$, $\bar{I}$, T_{low}, T_{high}, A, the list of segment traces $\{X_j\}$, and indices $\{k_j\}$.

5.3. Shared Memory Processor

A custom cross-platform processing module was developed to execute Python signal processing routines from within MATLAB, building upon the `shared_memory` module [54], with minor modifications documented on GitHub. The system comprised (1) a MATLAB interface for managing shared memory exchange, and (2) a Python backend for function execution.

A custom shared memory interface was developed to allow execution of Python signal processing routines from within MATLAB. The system initialized a shared memory session, exported input variables either as memory segments or inline JSON depending on data complexity, triggered the specified Python function, and retrieved the outputs for further analysis. Automatic transposition of row vectors ($1 \times N$) to column vectors ($N \times 1$) was performed during import to ensure MATLAB compatibility.

The Python backend, structured as the `shared_memory _processor` module, managed memory sessions, function registration, and data exchange. It depended on the external `shared_memory` package for low-level memory operations [54]. Registered functions were dynamically loaded at runtime, with inputs retrieved from shared memory, passed into the function, and outputs stored back for MATLAB retrieval.

Metadata for each session was maintained in a standardized `_metadata.json` file. Error checking was implemented to ensure consistency and memory integrity across function calls.

References

[1] Kumari, S.; Samara, M.; Ampadi Ramachandran, R.; Gosh, S.; George, H.; Wang, R.; Pesavento, R. P.; Mathew, M. T. A Review on Saliva-Based Health Diagnostics: Biomarker Selection and Future Directions. *Biomedical Materials & Devices (New York, N.y.)* **2023**, 1–18.

[2] Saville Melanie; Cramer Jakob P.; Downham Matthew; Hacker Adam; Lurie Nicole; Van der Veken Lieven; Whelan Mike; Hatchett Richard Delivering Pandemic Vaccines in 100 Days — What Will It Take? *New England Journal of Medicine* **2022**, *387*, e3.

[3] Meca-Lallana, J. E.; Casanova, B.; Rodriguez-Antiguedad, A.; Eichau, S.; Izquierdo, G.; Duran, C.; Rio, J.; Hernandez, M. A.; Calles, C.; Prieto-Gonzalez, J. M.; Ara, J. R.; Uria, D. F.; Costa-Frossard, L.; Garcia-Merino, A.; Oreja-Guevara, C. Consensus on early detection of disease progression in patients with multiple sclerosis. *Frontiers in Neurology* **2022**, *13*, 931014.

[4] Magri, V.; Marino, L.; De Renzi, G.; De Meo, M.; Salvatori, F.; Buccilli, D.; Bianco, V.; Santini, D.; Nicolazzo, C.; Gazzaniga, P. Early Detection of Disease Progression in Metastatic Cancers: Could CTCs Improve RECIST Criteria? *Biomedicines* **2024**, *12*, 388.

[5] Groenewegen, A.; Zwartkruis, V. W.; Rienstra, M.; Zuithoff, N. P. A.; Hollander, M.; Koffijberg, H.; Wolcherink, M. O.; Cramer, M. J.; Schouw, Y. T. v. d.; Hoes, A. W.; Rutten, F. H.; Boer, R. A. d. Diagnostic yield of a proactive strategy for early detection of cardiovascular disease versus usual care in adults with type 2 diabetes or chronic obstructive pulmonary disease in primary care in the Netherlands (RED-CVD): a multicentre, pragmatic, cluster-randomised, controlled trial. *The Lancet Public Health* **2024**, *9*, e88–e99.

[6] Chu, H.; Liu, C.; Liu, J.; Yang, J.; Li, Y.; Zhang, X. Recent advances and challenges of biosensing in point-of-care molecular diagnosis. *Sensors and Actuators B: Chemical* **2021**, *348*, 130708.

[7] Van Der Pol, B. Overview of point-of-care diagnostic options for detection of chlamydia trachomatis: current technology and implementation considerations. *Expert Review of Molecular Diagnostics* **2025**, *25*, 47–58, Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/14737159.2025.2453505.

[8] Jeyaraman, M.; Jeyaraman, N.; Ramasubramanian, S.; Balaji, S.; Iyengar, K. P.; Jain, V. K.; Rajendran, R. L.; Gangadaran, P. Nanomaterials in point-of-care diagnostics: Bridging the gap between laboratory and clinical practice. *Pathology - Research and Practice* **2024**, *263*, 155685.

[9] Goncharov, A.; Joung, H.-A.; Ghosh, R.; Han, G.-R.; Ballard, Z. S.; Maloney, Q.; Bell, A.; Aung, C. T. Z.; Garner, O. B.; Carlo, D. D.; Ozcan, A. Deep learning-enabled multiplexed point-of-care sensor using a paper-based fluorescence vertical flow assay. *Small* **2023**, *19*, 2300617, arXiv:2301.10934 [physics].

[10] Simonyan, V.; Mazumder, R. High-Performance Integrated Virtual Environment (HIVE) Tools and Applications for Big Data Analysis. *Genes* **2014**, *5*, 957–981, Number: 4 Publisher: Multidisciplinary Digital Publishing Institute.

[11] Naresh, V.; Lee, N. A Review on Biosensors and Recent Development of Nanostructured Materials-Enabled Biosensors. *Sensors (Basel, Switzerland)* **2021**, *21*, 1109.

[12] Murphy, A. C.; Wechsler, M. E.; Peppas, N. A. Recent advancements in biosensing approaches for screening and diagnostic applications. *Current Opinion in Biomedical Engineering* **2021**, *19*, 100318.

[13] Stuber, A.; Schlotter, T.; Hengsteler, J.; Nakatsuka, N. In *Trends in Biosensing Research: Advances, Challenges and Applications*; Lisdat, F., Plumeré, N., Eds.; Springer International Publishing: Cham, 2024; pp 283–316.

[14] Akbari Nakhjavani, S.; Mirzajani, H.; Carrara, S.; Onbaşlı, M. C. Advances in biosensor technologies for infectious diseases detection. *TrAC Trends in Analytical Chemistry* **2024**, *180*, 117979.

[15] Rao Bommi, J.; Kummari, S.; Lakavath, K.; Sukumaran, R. A.; Panicker, L. R.; Marty, J. L.; Yugender Goud, K. Recent Trends in Biosensing and Diagnostic Methods for Novel Cancer Biomarkers. *Biosensors* **2023**, *13*, 398, Number: 3 Publisher: Multidisciplinary Digital Publishing Institute.

[16] Xue, L.; Yamazaki, H.; Ren, R.; Wanunu, M.; Ivanov, A. P.; Edel, J. B. Solid-state nanopore sensors. *Nature Reviews Materials* **2020**, *5*, 931–951, Publisher: Nature Publishing Group.

[17] Wang, Y.; Zhao, Y.; Bollas, A.; Wang, Y.; Au, K. F. Nanopore sequencing technology, bioinformatics and applications. *Nature Biotechnology* **2021**, *39*, 1348–1365, Publisher: Nature Publishing Group.

[18] Coulter, W. H. Means for counting particles suspended in a fluid. 1953.

[19] Wen, C.; Zhang, S.-L. Fundamentals and potentials of solid-state nanopores: a review. *Journal of Physics D: Applied Physics* **2020**, *54*, 023001.

[20] Rahman, M.; Sampad, M. J. N.; Hawkins, A.; Schmidt, H. Recent advances in integrated solid-state nanopore sensors. *Lab on a chip* **2021**, *21*, 3030–3052.

[21] He, Y.; Tsutsui, M.; Zhou, Y.; Miao, X.-S. Solid-state nanopore systems: from materials to applications. *NPG Asia Materials* **2021**, *13*, 1–26, Publisher: Nature Publishing Group.

[22] Yan, H.; Chen, T.; Hu, G.; Ma, J.; Wu, L.; Tu, J. A long-term stable solid-state nanopore for the dynamic monitoring of DNA synthesis. *Analytica Chimica Acta* **2025**, *1344*, 343710.

[23] Liang, L.; Qin, F.; Wang, S.; Wu, J.; Li, R.; Wang, Z.; Ren, M.; Liu, D.; Wang, D.; Astruc, D. Overview of the materials design and sensing strategies of nanopore devices. *Coordination Chemistry Reviews* **2023**, *478*, 214998.

[24] Liang, S.; Xiang, F.; Tang, Z.; Nouri, R.; He, X.; Dong, M.; Guan, W. Noise in nanopore sensors: Sources, models, reduction, and benchmarking. *Nanotechnology and Precision Engineering* **2020**, *3*, 9–17.

[25] Wanunu, M. Nanopores: A journey towards DNA sequencing. *Physics of Life Reviews* **2012**, *9*, 125–158.

[26] Roelen, Z.; Briggs, K.; Tabard-Cossa, V. Analysis of Nanopore Data: Classification Strategies for an Unbiased Curation of Single-Molecule Events from DNA Nanostructures. *ACS Sensors* **2023**, *8*, 2809–2823, Publisher: American Chemical Society.

[27] Kowalczyk, S. W.; Grosberg, A. Y.; Rabin, Y.; Dekker, C. Modeling the conductance and DNA blockade of solid-state nanopores. *Nanotechnology* **2011**, *22*, 315101.

[28] Dekker, C. Solid-state nanopores. *Nature Nanotechnology* **2007**, *2*, 209–215, Publisher: Nature Publishing Group.

[29] Branton, D. et al. The potential and challenges of nanopore sequencing. *Nature Biotechnology* **2008**, *26*, 1146–1153, Publisher: Nature Publishing Group.

[30] Wen, C.; Dematties, D.; Zhang, S.-L. A Guide to Signal Processing Algorithms for Nanopore Sensors. *ACS Sensors* **2021**, *6*, 3536–3555, Publisher: American Chemical Society.

[31] Koch, C.; Reilly-O'Donnell, B.; Gutierrez, R.; Lucarelli, C.; Ng, F. S.; Gorelik, J.; Ivanov, A. P.; Edel, J. B. Nanopore sequencing of DNA-barcoded probes for highly multiplexed detection of microRNA, proteins and small biomarkers. *Nature Nanotechnology* **2023**, *18*, 1483–1491, Publisher: Nature Publishing Group.

[32] Ren, R. et al. Multiplexed detection of viral antigen and RNA using nanopore sensing and encoded molecular probes. *Nature Communications* **2023**, *14*, 7362, Publisher: Nature Publishing Group.

[33] Ren, R.; Sun, M.; Goel, P.; Cai, S.; Kotov, N. A.; Kuang, H.; Xu, C.; Ivanov, A. P.; Edel, J. B. Single-Molecule Binding Assay Using Nanopores and Dimeric NP Conjugates. *Advanced Materials* **2021**, *33*, 2103067, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/adma.202103067.

[34] Niazai, S.; Rahimzai, A. A.; Atifnigar, H. Applications of MATLAB in Natural Sciences: A Comprehensive Review. *European Journal of Theoretical and Applied Sciences* **2023**, *1*, 1006–1015, Number: 5.

[35] MathWorks Working with Large Data Sets. MathWorks, 2025; Accessed: 2025-04-11.

[36] Ziogas, A. N.; Schneider, T.; Ben-Nun, T.; Calotoiu, A.; Matteis, T. D.; Licht, J. d. F.; Lavarini, L.; Hoefler, T. Productivity, Portability, Performance: Data-Centric Python. *ACM Computing Surveys* **2022**, 1122445.1122456, arXiv:2107.00555 [cs].

[37] Fink, Z.; Liu, S.; Choi, J.; Diener, M.; Kale, L. V. Performance Evaluation of Python Parallel Programming Models: Charm4Py and mpi4py. 2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2). 2021; pp 38–44, arXiv:2111.04872 [cs].

[38] Shivashankar, K.; Martini, A. Better Python Programming for all: With the focus on Maintainability. *arXiv preprint arXiv:2408.09134* **2024**,

[39] Xi, G.; Su, J.; Ma, J.; Wu, L.; Tu, J. A robust signal processing program for nanopore signals using dynamic correction threshold with compatible baseline fluctuations. *Analyst* **2025**, *150*, 1386–1397, Publisher: Royal Society of Chemistry.

[40] Ni, P.; Huang, N.; Zhang, Z.; Wang, D.-P.; Liang, F.; Miao, Y.; Xiao, C.-L.; Luo, F.; Wang, J. DeepSignal: detecting DNA methylation state from Nanopore sequencing reads using deep-learning. *Bioinformatics* **2019**, *35*, 4586–4595.

[41] Dutt, S.; Shao, H.; Karawdeniya, B.; Bandara, Y. M. N. D. Y.; Daskalaki, E.; Suominen, H.; Kluth, P. High Accuracy Protein Identification: Fusion of Solid-State Nanopore Sensing and Machine Learning. *Small Methods* **2023**, *7*, 2300676, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smtd.202300676.

[42] Chau, C. C. C.; Weckman, N. E.; Thomson, E. E.; Actis, P. Solid-State Nanopore Real-Time Assay for Monitoring Cas9 Endonuclease Reactivity. *ACS Nano* **2025**, *19*, 3839–3851, Publisher: American Chemical Society.

[43] Johnson, J.; Galigekere, C. R.; Varma, M. M. A Solid-State Nanopore Signal Generator for Training Machine Learning Models. 2025; `http://arxiv.org/abs/2504.05466`, arXiv:2504.05466 [eess].

[44] Rosenstein, J. K.; Wanunu, M.; Merchant, C. A.; Drndic, M.; Shepard, K. L. Integrated nanopore sensing platform with sub-microsecond temporal resolution. *Nature Methods* **2012**, *9*, 487–492, Publisher: Nature Publishing Group.

[45] MathWorks movmean - Moving Average - MATLAB. `https://uk.mathworks.com/help/matlab/ref/movmean.html`, 2024; Accessed: 2025-04-11.

[46] Whittaker, E. T. On a New Method of Graduation. *Proceedings of the Edinburgh Mathematical Society* **1922**, *41*, 63–75.

[47] Eilers, P. H. C. A Perfect Smoother. *Analytical Chemistry* **2003**, *75*, 3631–3636, Publisher: American Chemical Society.

[48] Erb, D. pybaselines: A Python library of algorithms for the baseline correction of experimental data. `https://github.com/derb12/pybaselines`.

[49] Seth, S.; Bhattacharya, A. How capture affects polymer translocation in a solitary nanopore. *The Journal of Chemical Physics* **2022**, *156*, 244902, arXiv:2111.11353 [cond-mat].

[50] The MathWorks, Inc. accumarray. The MathWorks, Inc.: Natick, Massachusetts, United States, 2025; Accessed: April 25, 2025.

[51] The MathWorks, Inc. pyenv. 2024; Accessed: 2025-04-21.

[52] The h5py Developers h5py: HDF5 for Python. 2024; Accessed: 2025-04-21.

[53] Python Software Foundation multiprocessing.shared_memory — Shared memory for direct access across processes. 2024; Accessed: 2025-04-21.

[54] Izd, B. Shared Memory: Share data between Python - Julia - Matlab. `https://github.com/ben-izd/shared_memory`, 2022; MIT License. Accessed: 2025-04-21.

[55] Virtanen, P. et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* **2020**, *17*, 261–272.

[56] Lam, S. K.; Pitrou, A.; Seibert, S. Numba: a LLVM-based Python JIT compiler. Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. New York, NY, USA, 2015; pp 1–6.

[57] Harden, S. W. pyABF: A Python library for reading Axon Binary Format (ABF) files. `https://swharden.com/pyabf/`, 2023; Version 2.3.8, released August 2, 2023.

[58] Kern, R.; contributors line_profiler: Line-by-line profiling for Python. `https://pypi.org/project/line-profiler/`, 2025; Accessed: 2025-04-29.

Supplementary Information

This section contains the core code developed in this work, divided into: Data Acquisition, Optimized Python Functions, Shared Memory Module (MATLAB), and Shared Memory Module (Python).

Full docstrings and the full project organization are available in the associated GitHub repository: `https://github.com/brandonhuang-projects/msc-project`. Readers are referred there for a complete description of function behavior, code structure, and usage.

All code presented here corresponds to the final working version used during benchmarking and integration testing and begins in the following pages.

Supplementary Information: Data Acquisition

Timer.m

```matlab
function Timer(pre_script, main_script, post_script, repeat)
    % Function to test the timing of a main script with setup and cleanup.

    temp_workspace_file = 'temp_workspace.mat';

    % Run pre-setup
    if ~isempty(pre_script)
        fprintf('-- Running pre-setup script: %s\n', pre_script);
        run(pre_script);

        % Save workspace
        save(temp_workspace_file);
    end

    fprintf('\n')

    timings = zeros(1, repeat, 'double');
    for i = 1:repeat

        fprintf('-- Timing main script: %s (Run %d of %d)\n', main_script, i, repeat);

        % Reload workspace
        load(temp_workspace_file);

        t0 = tic;
        run(main_script);
        timings(i) = toc(t0);
    end

    fprintf('\n');

    % Calculate mean and standard deviation
    mean_time = mean(timings);
    std_dev_time = std(timings);

    % Display the timing result
    fprintf('Elapsed time (MATLAB): %.6f s +/- %.6f s (mean +/- std. dev. of %d runs)\n', ...
        mean_time, std_dev_time, repeat);

    % Run post-cleanup
    if ~isempty(post_script)
        run(post_script);
    end

end
```

Timer_py.py

```python
import time as time_import
import statistics

import pyabf # ABF File Handling
import warnings

# Suppress all warnings
warnings.filterwarnings("ignore")

from line_profiler import LineProfiler


```

```python
print("Imports Initialized")

# ----- Timer Functions -------------------------

global_kwargs = None

def time_python(pre_function, function, post_function, repeat):

    global global_kwargs

    time_list = [] # Store execution times
    result = None # Output placeholder

    # Pre-script setup
    print('-- Running pre-setup script...')

    if not global_kwargs:
        print("-- Importing...")
        global_kwargs = pre_function()
        global_kwargs['timer'] = True

    # Per designated repeats:
    for i in range(0, repeat + 1):
        print(f'-- Timing main function (Run {i} of {repeat})', flush=True)
        start_time = time_import.time() # Start timer
        result = function(**global_kwargs) # Run main function
        end_time = time_import.time() # End timer

        if i != 0: # Flush first repeat due to pre_function overhead
                   # causing overestimations in runtime
            time_list.append(end_time - start_time)
            print(f'Elapsed time is {round(time_list[-1],6)} seconds.')

        else:
            print(f'Elapsed time flushed for first run')
        post_function()

    # Mean and standard deviation
    mean_time = statistics.mean(time_list)
    std_dev_time = statistics.stdev(time_list)

    print(f"Elapsed time (Python): {mean_time:.6f} s +/- {std_dev_time:.6f} s") #(mean +/- std. dev. of {repeat} runs)")


def run_line_profile(function,**kwargs):
    """
    Profiles the line-by-line execution time of a given function.
    """
    kwargs['timer'] = True
    lp = LineProfiler()
    lp_wrapper = lp(function)
    lp_wrapper(**kwargs)
    return lp.print_stats()
```

Supplementary Information: Optimized Python Functions

status.py

```python
# custom_functions/status.py
import numpy as np

def status(data1, data2, data3, data4, data5, data6, data7, data8, data9):

    out1 = data1.upper() if isinstance(data1, str) else data1
    out2 = np.mean(data2)
    out3 = np.sum(data3)
    out4 = np.max(data4)
    out5 = np.sum(data5)
    out6 = np.conjugate(data6)
    out7 = np.round(data7,2)
    out8 = [data8[:2], data8[2:]]
    out9 = {
        'a': data9['a'] + 1,
        'b': data9['b'].upper(),
        'c': float(np.sum(data9['c'])),
        'd': not data9['d']
        }

    return out1, out2, out3, out4, out5, out6, out7, out8
```

resample.py

```python
# custom_functions/resample.py

import numpy as np
from scipy.signal import resample_poly

def resample(Count_2, num, denom, padding=10):

    # Compute the mean of the first and last 10 points
    start_mean = np.mean(Count_2[:10])
    end_mean = np.mean(Count_2[-11:])

    # Create a padded signal using the computed boundary means
    padded_signal = np.concatenate([
        np.full((padding,), start_mean),
        Count_2,
        np.full((padding,), end_mean)
    ])

    # Perform resampling with specified up/down factors
    resampled_signal = resample_poly(padded_signal, up=num, down=denom)

    # Remove padding and return the final resampled signal
    return resampled_signal[padding:-padding]
```

movmean.py

```python
# custom_functions/movmean.py
import numpy as np
from numba import njit

@njit
def movmean_jit(A, k):

    n = A.shape[0]

    # If window size is 1, return a copy of the input array
    if k == 1:
        return A.copy()

    # Compute cumulative sum with a prepended zero
    cumsum = np.empty(n + 1, dtype=A.dtype)
    cumsum[0] = 0.0
    for i in range(n):
        cumsum[i + 1] = cumsum[i] + A[i]

    # Compute moving mean for fully available windows
    mlen = n - k + 1
    M_core = np.empty(mlen, dtype=A.dtype)
    for i in range(mlen):
        M_core[i] = (cumsum[i + k] - cumsum[i]) / k

    # Boundary window sizes
    f = k // 2 # Points at the start
    e = (k - 1) // 2 # Points at the end

    # Prepare output array
    out = np.empty(n, dtype=A.dtype)

    # Front boundary: growing window
    for i in range(f):
        s = 0.0
        for j in range(i + 1):
            s += A[j]
        out[i] = s / (i + 1)

    # Core: centered full windows
    for i in range(mlen):
        out[f + i] = M_core[i]

    # End boundary: shrinking window
    for i in range(e):
        s = 0.0
        for j in range(n - (i + 2), n):
            s += A[j]
        out[f + mlen + i] = s / (i + 2)

    return out


def movmean(Count_2, MovMean_Const):

    Base = movmean_jit(Count_2, MovMean_Const)

    # Lower baseline by 0.75 times standard deviation
    Base = Base - (0.75 * Base.std(ddof=1))

    # Subtract baseline from original signal
    CountBase = Count_2 - Base

    # Set negative values to zero
    CountBase = np.maximum(CountBase, 0)

    return CountBase
```

whittaker.py

```
# custom_functions/whittaker.py

from pybaselines import Baseline

def whittaker(y, lam, p, order):
    # Perform asymmetric least squares baseline correction
    Base, _ = Baseline().asls(y, lam=lam, p=p, diff_order=order, max_iter=10)
    return Base
```

threshold.py

```
# custom_functions/threshold.py
import numpy as np
from scipy.optimize import least_squares
from scipy.stats import poisson
from scipy.special import gamma
from math import sqrt
from numba import njit

@njit
def fast_histogram(data, bins, step):
    hist = np.zeros(len(bins), dtype=np.int64) # No len(bins)-1 since MATLAB includes

    for value in data:
        bin_index = int(value / step) # Convert int for index
        if 0 <= bin_index < len(hist):
            hist[bin_index] += 1

    return hist

# Poission PMF
def Pois(u, x):
    return (u**x) * np.exp(-u) / gamma(x + 1)

def threshold(CountBase, STD, SO, ThrStep):

    # Calculate the first difference
    Df = np.diff(CountBase)
    # Replace any NaN values 0
    Df[np.isnan(Df)] = 0

    # Compute mean of the absolute deltas
    Step = np.mean(abs(Df))

    # Set minimum step value
    if Step < 0.0001:
        Step = 0.001

    max_val = np.max(CountBase)

    StepVec = np.arange(0,max_val/2,Step)

    # Refine StepVec if too small
    if len(StepVec) < 25:
        Step = Step/2
        StepVec = np.arange(0,max_val/2,Step)

    # Override Threshold
    if ThrStep['Overide'] == True:
        Step = ThrStep['Step']
        MX = ThrStep['MX']
        StepVec = range(0,MX,Step)

    # CountBase histogram with StepVec bins
    Hist = fast_histogram(CountBase, StepVec, Step)

    # Set curve fit upper bound; inf if small
    if len(CountBase) > 1000:
        UL = (np.mean(CountBase[0:1000])/Step)*10
    else:
        UL = float('inf')

    # Normalize
    Hist = Hist/np.sum(Hist) # MATLAB includes 0 at end after histc()

    # Remove first element
    Hist = Hist[1:]
    StepVec = StepVec[1:]

    pois_array = np.arange(1, len(StepVec) + 1)

    # Least squares curve fitting | X ~ Pois(lambda)
    # (2) Adapted from https://uk.mathworks.com/help/optim/ug/lsqcurvefit.html#buuhcjo-3
    try:
        # least-squares with discrete integer indieces
        result = least_squares(lambda u: Pois(u, pois_array) - Hist,
                               x0 = 5, bounds = (0, UL), method = 'trf',
                               ftol=1e-6, xtol=1e-6, gtol=1e-6, diff_step=1e-6) # as per (2)
                               #max_nfev = (100 * len(StepVec)), ) # as per (2)
    except:
        result = least_squares(lambda u: Pois(u, pois_array) - Hist,
                               x0 = 2, bounds = (0, UL), method = 'trf',
                               ftol=1e-6, xtol=1e-6, gtol=1e-6, diff_step=1e-6) # as per (2)
                               #max_nfev = (100 * len(StepVec)), ) # as per (2)
    PoisLamda = result.x[0]

    # Compute fitted Poisson PMF
    PoisFit = poisson.pmf(np.arange(1, len(StepVec) + 1), PoisLamda)

    # Normalize to histogram scale
    PoisFit = PoisFit / (np.max(PoisFit) * np.max(Hist))

    thresh = PoisLamda + (STD * sqrt(PoisLamda))
    thresh = thresh * Step
    PoisLamda = PoisLamda * Step * SO

    return thresh, PoisLamda, StepVec, Hist, PoisFit
```

peakfinder.py

```
# custom_functions/peakfinder.py
import numpy as np
from functools import wraps
from collections import defaultdict


## - - - - - Utilities - - - - - ##

def list_to_np_array(func):
    @wraps(func)
    def wrapper(*args, **kwargs):

        # Convert all list-type arguments to np.array
        new_args = [np.array(arg) if isinstance(arg, list) else arg for arg in args]
        new_kwargs = {k: np.array(v) if isinstance(v, list) else v for k, v in kwargs.items()}

        return func(*new_args, **new_kwargs)

    return wrapper

@list_to_np_array
def convert_to_numpy(*args):
    # To convert list to arrays
    return args

def clean_float64_output(func):
```

```python
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Call the original function
        result = func(*args, **kwargs)

        # Convert the result to a NumPy array of dtype float64
        try:
            result = np.array(result, dtype=np.float64)
        except Exception as e:
            pass
        return result
    return wrapper

## - - - - - Helper Functions - - - - - ##

def strfind(arr,subarray):
    for i in range(len(arr) - len(subarray) + 1):
        if arr[i : i + len(subarray)] == subarray:
            return i
    return -1

def accumarray(ind, data, func = np.sum):
    # !! ASSUMES IND IS SORTED and 1-based indexed!!

    ind = np.array(ind) - 1 # Convert 1-based indexing to 0-based

    sz = np.max(ind) + 1 # Define size based on the maximum index

    # Sum data by index
    if func == np.sum:

        # np.bincount for efficient summation based on index array
        return np.bincount(ind, weights=data, minlength=sz)

    # Calculate mean by index
    elif func == np.mean:

        # Uses np.bincount to get sum and count per index
        sum_array = np.bincount(ind, weights=data, minlength=sz)
        count_array = np.bincount(ind, minlength=sz)

        # Divides to find the mean; zero-handling via pre-initialized array
        # and subsequent restriction
        return np.divide(sum_array, count_array,
                         out=np.zeros_like(sum_array, dtype=float),
                         where=(count_array > 0))

    # Calculate max value by index
    # O(n) complexity
    elif func == np.max:

        # Initialize with lowest index
        output = np.full(sz, 0.0)

        for idx, value in zip(ind, data):

            # Update if current value is greater
            if value > output[idx]:
                output[idx] = value

        return output

    # Retrieve first item by index
    # O(n) complexity
    elif func == (lambda x: x[0]):
        output = np.zeros(sz) # Initialize output
        current_index = -1 # Track index

        # First occurrence only
        for idx, value in zip(ind, data):
            if idx != current_index:
                output[idx] = value
                current_index = idx

        return output

    # Retrieve last item by index
    # O(n) complexity
    elif func == (lambda x: x[-1]):
        output = np.zeros(sz) # Initialize output

        # Overwriting so the last occurrence is kept
        for idx, value in zip(ind, data):
            output[idx] = value

        return output

    # Apply other functions on accumulated lists
    else:
        output = [None] * sz # Initialize list output
        accumulator = defaultdict(list) # Store data by index via defaultdict

        # Accumulate data into lists for each index
        for i, idx in enumerate(ind):
            accumulator[idx].append(data[i])

        # Apply function on accumulated lists for each index
        for idx, values in accumulator.items():
                output[idx] = func(values)

        try:
            # Convert to numpy array; replace None with 0
            return np.array([x if x is not None else 0 for x in output])
        except:
            try:
                # Return object array if invalid
                return np.array([x if x is not None else 0 for x in output], dtype=object)

            except Exception as e:
                # Return as list
                return [x if x is not None else 0 for x in output]

@list_to_np_array
def findTime(indx, s2, time):

# !! Relies on 0-based indexing of indx
# Replicates purpose, not data fom MATLAB original

    subset = s2[indx]
    ix_loc = np.argmax(subset)

    ix = [ix_loc + 1, time[indx[ix_loc]]]

    return ix
```

```

## - - - - - Main Functions - - - - - ##

def FindPeaks_V2(CountBase, PoisLamda, thresh,
               time, T_res, FullWidthHM, WidthLimit,
               CurrentLimit, FileCondition, Buff):

    # Call PeaksBeta_V2
    TiMaxBurst, PkMaxBurst, MeanBurst, TiLow, TiHigh, Area, TEST, PeakIndex = \
        PeaksBeta_V2(CountBase, PoisLamda, thresh, time, Buff)

    # Filter out invalid bursts based on conditions
    ValZero = (TiHigh - TiLow) > 0
    TiMaxBurst = TiMaxBurst[ValZero]
    PkMaxBurst = PkMaxBurst[ValZero]
    MeanBurst = MeanBurst[ValZero]
    TiLow = TiLow[ValZero]
    TiHigh = TiHigh[ValZero]
    Area = Area[ValZero]
    TEST = TEST[ValZero] #[row for row, include in zip(TEST, ValZero) if include]
    PeakIndex = PeakIndex[ValZero]

    if WidthLimit['Low_on'] == 1:
        ValC = (TiHigh - TiLow) >= WidthLimit['Low_with']
        TiMaxBurst = TiMaxBurst(ValC);
        PkMaxBurst = PkMaxBurst(ValC);
        MeanBurst = MeanBurst(ValC);
        TiLow = TiLow(ValC);
        TiHigh = TiHigh(ValC);
        Area = Area(ValC);
        TEST = TEST[ValZero]
        PeakIndex = PeakIndex(ValC);


    if WidthLimit['Upper_on'] == 1:
        ValC = (TiHigh - TiLow) <= WidthLimit['Upper_with']
        TiMaxBurst = TiMaxBurst(ValC);
        PkMaxBurst = PkMaxBurst(ValC);
        MeanBurst = MeanBurst(ValC);
        TiLow = TiLow(ValC);
        TiHigh = TiHigh(ValC);
        Area = Area(ValC);
        TEST = TEST[ValZero]
        PeakIndex = PeakIndex(ValC);


    if CurrentLimit['on'] == 1:
        # PkMaxBurst
        ValD = (PkMaxBurst <= CurrentLimit['Value']/1000) # convert to nA
        TiMaxBurst = TiMaxBurst(ValD)
        PkMaxBurst = PkMaxBurst(ValD)
        MeanBurst = MeanBurst(ValD)
        TiLow = TiLow(ValD)
        TiHigh = TiHigh(ValD)
        Area = Area(ValD)
        TEST = TEST[ValZero]
        PeakIndex = PeakIndex(ValD)

    PeakIndexRaw = np.round(TiMaxBurst/T_res)


    if FullWidthHM['On'] == 1: # full width half max of bursts
```

```
        # For each burst:

        MX = [] # Max value
        MX_Loc = [] # Index of max value
        Y = [] # Half-maximum threshold
        p1 = [] # Left boundary index of FWHM
        p2 = [] # Relative right boundary index
        p3 = [] # Absolute right boundary index (for whole burst)

        PEAKS_FWHM = [] # Burst segments
        AreaFWHM = [] # AUC (sum)

        # Iterating over bursts
        for X in TEST:
            max_value = np.max(X)
            max_index = np.argmax(X)

            MX.append(max_value) # Append max value to MX list
            MX_Loc.append(max_index) # Append index of max value to MX_Loc list

            # Step 2: Calculate the half-maximum threshold
            half_max = max_value / FullWidthHM['factor']
            Y.append(half_max)

            left_index = np.argmin(abs(X[:max_index + 1] - half_max))
            p1.append(left_index)

            right_index = np.argmin(abs(X[max_index:] - half_max))
            p2.append(right_index)

            absolute_right_index = min(right_index + max_index + 1, len(X) - 1)
            p3.append(absolute_right_index)

            fwhm_segment = X[left_index:absolute_right_index + 1]
            PEAKS_FWHM.append(fwhm_segment)

            auc = (2 * np.sum(fwhm_segment) - fwhm_segment[0] - fwhm_segment[-1]) / 2
            AreaFWHM.append(auc)

        p1, p2, p3, MX, MX_Loc = convert_to_numpy(p1, p2, p3, MX, MX_Loc) # as such

        TEST = PEAKS_FWHM

        Area = np.array(AreaFWHM)

        WidthFWHM = (p3-p1) * T_res

        TiLowFWHM = TiMaxBurst + ((p1 - MX_Loc) * T_res)
        TiHighFWHM = TiMaxBurst + ((p3 - MX_Loc) * T_res)

        TiLow = TiLowFWHM
        TiHigh = TiHighFWHM

        PeakIndex = (PeakIndex - MX_Loc + (p3 - p1)/2+1);

    if WidthLimit['Low_on'] == 1:
        ValC = (TiHigh - TiLow) >= WidthLimit['Low_width']
        TiMaxBurst = TiMaxBurst(ValC)
        PkMaxBurst = PkMaxBurst(ValC)
        MeanBurst = MeanBurst(ValC)
        TiLow = TiLow(ValC)
        TiHigh = TiHigh(ValC)
        Area = Area(ValC)
        TEST = TEST[ValZero]
        PeakIndex = PeakIndex(ValC)

    return TiMaxBurst, PkMaxBurst, MeanBurst, TiLow, TiHigh, Area, TEST, PeakIndex

@clean_float64_output
def PeaksBeta_V2(Count, PoisLamda, thresh, time_raw, Buff):

    s = np.array(Count)
    #s += np.random.rand(len(s)) * 1e-9
    Data = s

    if Buff['On'] == 1:
        g = np.array(s)
        g[g < PoisLamda] = 0
        g[g >= PoisLamda] = 1

        for j in range(Buff['Numb']):
            r = strfind(g,[1,1,1] + [0]*j + [1])
            for n in range(j):
                g[r+3+n-2] = 1 # Adjust for 0-based indexing

            r = strfind([1] + [0]*j + [1,1,1])
            for n in range(j):
                g[r+1+n-2] = 1 # Adjust for 0-based indexing

        s[g == 0] = 0 # Zero elements in s where g is 0

    else:
        s[s < PoisLamda] = 0 # Zero elements below PoisLamda

    p = (s != 0).astype(int) # 1 if s is non-zero else 0

    # Detect and transform bursts in p
    ps = np.concatenate(([p[0]], np.diff(p, axis = 0))) # Detect transitions in p
    ps[ps != 1] = 0 # Marks transition
    ps = np.cumsum(ps) * p # Unique labels for each sequence

    # Keep non-zero values
    s = s[s != 0]
    #g = g[g != 0]
    p = ps[ps != 0]

    m = accumarray(p, s, func = np.max)

    # - - - - - - - - - - -

    FndAboveThresh = np.where(np.atleast_1d(m) >= thresh)[0] # atleast_1d() for consistency
    ind_new = np.isin(ps, FndAboveThresh + 1) # !! Accommodate for 1-indexing of ps

    s2 = Data[ind_new]
    time = np.array(time_raw)[ind_new]

    # Identify start points of new bursts
    ind_new2 = np.concatenate(([ind_new[0]], np.diff(ind_new.astype(int))))
    ind_new2 = np.where(ind_new2 != 1, 0, ind_new2)

    ind_start = ind_new2 # Save start markers

    # Cumulative sums of burst indicators
    ind_new2 = np.cumsum(ind_new2) * ind_new
```

```python
    ind_new3 = ind_new2[ind_new2 != 0]

    first = accumarray(ind_new3, s2, (lambda x: x[0])) # First item in each burst
    last = accumarray(ind_new3, s2, (lambda x: x[-1])) # Last item in each burst
    MeanBurst = accumarray(ind_new3, s2, np.mean) # Mean value for each burst
    SUM = accumarray(ind_new3, s2, np.sum) # Total sum for each burst

    # Collect burst segment as an array
    TEST = accumarray(ind_new3, s2, (lambda x: np.array(x))) # SLOW; no need for [x] call

    # Find peak time for each burst segment
    Output = accumarray(ind_new3, range(len(s2)), (lambda x: findTime(x, s2, time)))
    # Second arg adjusted for 0-based indexing

    try:
        Output = np.array(Output)
    except:
        pass

    # Calculate the AUC for burst
    Area = (2 * SUM - first - last)/2

    TiMaxBurst = Output[:,1] # Extract peak time
    PkMaxBurst = m[FndAboveThresh] # Extract peak value

    TiLow = accumarray(ind_new3, time, (lambda x: x[0])) # Start time of each burst
    TiHigh = accumarray(ind_new3, time, (lambda x: x[-1])) # End time of each burst

    ind_start = np.where(np.atleast_1d(ind_start) == 1)[0] + Output[:,0]
    PeakIndex = ind_start+1

    return TiMaxBurst, PkMaxBurst, MeanBurst, TiLow, TiHigh, Area, TEST, PeakIndex

def peakfinder(CountBase, PoisLamda, thresh, time, T_res, FullWidthHM,
               WidthLimit, CurrentLimit, FileCondition, Buff):

    TiMaxBurst, PkMaxBurst, MeanBurst, TiLow, TiHigh, Area, Event_all, PeakIndex = \
        FindPeaks_V2(CountBase, PoisLamda, thresh, time, T_res, FullWidthHM,
                     WidthLimit, CurrentLimit, FileCondition, Buff)

    return TiMaxBurst, PkMaxBurst, MeanBurst, TiLow, TiHigh, Area, Event_all, PeakIndex
```

Supplementary Information: Shared Memory Module (MATLAB)

shmemStatus.m

```matlab
function outcome = shmemStatus()
    addpath(fullfile(fileparts(mfilename('fullpath')),'shared_memory_processor'))
    func_names = initProcessor();

    data1 = "Hello, Shared Memory!"; % String
    data2 = [1.1, 2.2, 3.3, 4.4]; % Numeric double array (np.float64)
    data3 = int8([1, -2, 3, -4]); % int8 array
    data4 = int16([100, -200, 300, -400]); % int16 array
    data5 = uint8([10, 20, 30, 40]); % uint8 array
    data6 = [1+2i, 3+4i, 5+6i]; % Complex double array
    data7 = 3.1415926535;
    data8 = [1,2,3,4,5];
    data9 = struct('a', 10, 'b', 'hello', 'c', [1,2,3], 'd', true);

    [out1, out2, out3, out4, out5, out6, out7, out8, out9] = runProcessor('status', ...
        data1, data2, data3, data4, data5, data6, data7, data8, data9);

    check = {'HELLO, SHARED MEMORY!', 2.75, -2, 300, 100, ([1-2i, 3-4i, 5-6i].'), 3.14, ...
        {[1;2];[3;4;5]}, struct('a', 11, 'b', 'HELLO', 'c', 6, 'd', false);};

    outcome = isequal(check,{out1, out2, out3, out4, out5, out6, out7, out8, out9});
```

createSharedMemorySegments.m

```
function metadata = createSharedMemorySegments(memoryDir, varargin)

    % Ensure the memory directory exists; create it if needed.
    if ~exist(memoryDir, 'dir')
        mkdir(memoryDir);
    end

    % Metadata file path
    metaFilePath = fullfile(memoryDir, '_metadata.json');

    % Load existing metadata if it exists
    if exist(metaFilePath, 'file')
        fid = fopen(metaFilePath, 'r');
        rawJson = fread(fid, '*char')';
        fclose(fid);
        metadata = jsondecode(rawJson);
    else
        error("Metadata file not found: %s. Ensure Python has initialized the session.", metaFilePath);
    end

    % Keep session ID unchanged
    sessionId = metadata.session_id;

    % Prepare a structure to hold metadata for all segments.
    metadata.segments = {};

    % Loop over each input data array.
    for i = 1:numel(varargin)

        data = varargin{i};
        varId = lower(dec2hex(randi([0, 2^32-1], 1),8));

        segMeta = struct();

        if isnumeric(data) && isscalar(data) && isreal(data)
            segMeta.varId = varId;
            segMeta.source = 'MATLAB';
            segMeta.inline = true;
            segMeta.inlineData = data;
            segMeta.dataType = class(data);
            segMeta.dimensions = [];
            segMeta.timestamp = char(datetime('now', 'Format', 'yyyy-MM-dd''T''HH:mm:ss'));

        else
            segmentFile = fullfile(memoryDir, [varId,'.shmem']);
            set_shared_memory_path(segmentFile);

            % For complex types, JSON-encode the data.
            if shouldSerialize(data)
                data = jsonencode(data);
                segDataType = 'json';
                segDimensions = [];
            else
                segDataType = class(data);
                segDimensions = size(data);
            end
```

```matlab

            % Write the data to the shared memory segment.
            set_shared_memory_data(data);

            % Collect metadata for this segment.
            segMeta.varId = varId;
            segMeta.source = 'MATLAB';
            segMeta.filePath = segmentFile;
            segMeta.dataType = segDataType;
            segMeta.dimensions = segDimensions;
            segMeta.timestamp = char(datetime('now', 'Format', 'yyyy-MM-dd''T''HH:mm:ss'));
        end

        % Append the segment metadata.
        metadata.segments{end+1} = segMeta;

    end

    % Write the JSON string to a metadata file inside the memory directory.
    metadata.session_id = sessionId;
    fid = fopen(metaFilePath, 'w');
    fprintf(fid, '%s', jsonencode(metadata, 'PrettyPrint', true));
    fclose(fid);

end
```

readSharedMemorySegments.m

```matlab
function varargout = readSharedMemorySegments(bufferPath)

    metaPath = fullfile(bufferPath, '_metadata.json');

    % Check that the metadata file exists.
    if exist(metaPath, 'file') ~= 2
        error('Metadata file not found: %s', metaPath);
    end

    % Read the JSON file.
    fid = fopen(metaPath, 'rt'); % Open in text mode
    if fid == -1
        error('Could not open metadata file: %s', metaPath);
    end
    jsonStr = fread(fid, '*char')';
    fclose(fid);

    metadata = jsondecode(jsonStr);
    if ~isfield(metadata, 'segments')
        error('Metadata file is missing the "segments" field.');
    end

    segments = metadata.segments;

    % Handle empty metadata case
    if isempty(segments)
        varargout = {};
        return;
    end

    % Prepare a cell array to hold the outputs.
    dataList = cell(numel(segments), 1);
    c = 0;

    % Loop over each segment.
    for i = 1:numel(segments)
        if iscell(segments)
            seg = segments{i};
        else
            seg = segments(i);
        end

        if isfield(seg, 'source') && strcmp(seg.source, 'MATLAB')
            continue; % Skip this iteration and move to the next one
        end

        % Use inline data if available.
        if isfield(seg, 'inline') && seg.inline

            c = c + 1;
            dataList{c} = seg.inlineData;
            continue;
        end

        % Retrieve variable name and file path.
        filePath = seg.filePath;

        % Set the shared memory path for this segment.
        set_shared_memory_path(filePath);

        % Retrieve the data.
        data = get_shared_memory_data();

        % If the data was stored as JSON, decode it.
        if isfield(seg, 'dataType') && strcmp(seg.dataType, 'json')
            data = jsondecode(data);
        end

        % Convert 1xN to Nx1 if necessary
        if isvector(data) && size(data,1) == 1 && size(data,2) > 1
            data = data.'; % transpose to column vector
        end

        % Store the data in the map using the variable name as the key.
        c = c + 1;
        dataList{c} = data;
    end

    % Preallocate varargout to the actual number of valid entries.
    varargout = cell(1, c);
    for j = 1:length(dataList)
        varargout{j} = dataList{j};
    end
end
```

initProcessor.m

```
function func_names = initProcessor()

    global SharedMemoryProcessor

    SharedMemoryProcessor = struct;

    SharedMemoryProcessor.MATLAB_PATH = fileparts(mfilename('fullpath'));

    SharedMemoryProcessor.DIR= fileparts(fileparts(mfilename('fullpath')));

    SharedMemoryProcessor.WINPYTHON_PATH = fullfile(SharedMemoryProcessor.DIR, 'WPy64-31190b5', 'python-3.11.9.amd64',
        'python.exe');
    SharedMemoryProcessor.pe = pyenv('Version', SharedMemoryProcessor.WINPYTHON_PATH);

    SharedMemoryProcessor.SHARED_MEMORY_PATH = fullfile(SharedMemoryProcessor.DIR, 'shared_memory-main');
    addpath(SharedMemoryProcessor.SHARED_MEMORY_PATH);
    addpath(fullfile(SharedMemoryProcessor.SHARED_MEMORY_PATH, 'matlab'));
    SharedMemoryProcessor.BUFFER_PATH = fullfile(SharedMemoryProcessor.DIR, 'shared_memory_processor', '__ipc_buffer__');

    insert(py.sys.path, int32(0), SharedMemoryProcessor.DIR);
    SharedMemoryProcessor.module = py.importlib.import_module('shared_memory_processor');

    SharedMemoryProcessor.functions = py.shared_memory_processor.Functions().get_function_list;

    func_names = string(py.list(SharedMemoryProcessor.functions.keys()));

    SharedMemoryProcessor.processor = py.shared_memory_processor.SHMEM_Processor();
```

runProcessor.m

```
function varargout = runProcessor(func, varargin)

    global SharedMemoryProcessor

    SharedMemoryProcessor.processor.clear_vars();

    % Initialize cell arrays for names and data

    for k = 1:numel(varargin)
        if islogical(varargin{k}) && isequal(size(varargin{k}), [1,1])
            varargin{k} = double(varargin{k});
        end
    end

    SharedMemoryProcessor.metadata = createSharedMemorySegments(SharedMemoryProcessor.BUFFER_PATH, varargin{:});

    if ischar(func) || isstring(func)
        func = SharedMemoryProcessor.functions{char(func)};
    end

    SharedMemoryProcessor.processor.call(func)

    [varargout{1:nargout}] = readSharedMemorySegments(SharedMemoryProcessor.BUFFER_PATH);

    for k = 1:nargout
        if isrow(varargout{k}) && numel(varargout{k}) > 1
            if ~(isstring(varargout{k}) || ischar(varargout{k}))
                varargout{k} = varargout{k}.';
            end
        end
    end
```

shouldSerialize.m

```
function flag = shouldSerialize(value)
% should_serialize determines whether the given value should be JSON encoded.
    if isnumeric(value) || ischar(value) || isstring(value)
        flag = false;
    elseif iscell(value)
        try
            % Attempt to convert the cell array to a matrix.
            cell2mat(value);
            flag = false;
        catch
```

```
            flag = true;
        end
    elseif isstruct(value)
        flag = true;
    else
        flag = true;
    end
end
```

Supplementary Information: Shared Memory Module (Python)

__init__.py

```
# shared_memory_processor/__init__.py
from .processor import SHMEM_Processor
from .decorators import classproperty, sync_shared_memory
from .function_registry import Functions, import_functions
from .utils import get_script_dir

__all__ = [
    "SHMEM_Processor",
    "classproperty",
    "sync_shared_memory",
    "Functions",
    "import_functions",
    "get_script_dir"
]

import os
current_dir = os.path.dirname(get_script_dir())
custom_functions_dir = os.path.join(current_dir, "custom_functions")

# Import and register functions
import_functions(custom_functions_dir, Functions, method_type='staticmethod')
```

decorators.py

```
# shared_memory_processor/decorators.py
from functools import wraps
import numpy as np
from .utils import should_serialize

class classproperty:
    """
    Descriptor for defining properties on the class itself.

    Allows methods to be accessed as class-level properties.
    """
    def __init__(self, fget):
        self.fget = fget # Store the getter function
    def __get__(self, instance, owner):
        # Call the getter with the class as the argument
        return self.fget(owner)

def sync_shared_memory(processor, func):
    """
    Decorator to synchronize shared memory with function calls.

    Reads shared memory data via the processor, processes it,
    passes it as arguments to the wrapped function, then updates
    the shared memory with the function's result.

    """
    @wraps(func)
```

```python
    def wrapper(*args, **kwargs):

        # Retrieve shared memory data using the processor
        shared_memory_data = processor._read_shared_memory_metadata()

        shared_memory_data = clean_input(shared_memory_data)

        # Call the original function with shared memory data
        result = func(*shared_memory_data)

        result = clean_output(result)

        # Update the shared memory segments with the processed result
        processor._create_shared_memory_segments(*result)

        return result
    return wrapper

def clean_input(shared_memory_data):

    for index, item in enumerate(shared_memory_data):

        # Remove singleton dimensions.
        if hasattr(item, 'squeeze'):
            try:
                item = item.squeeze()
            except Exception:
                pass

        # Convert it to a native Python scalar
        if hasattr(item, 'item'):
            try:
                item = item.item()
            except Exception:
                pass

        shared_memory_data[index] = item

    return shared_memory_data

def clean_output(result):
    # Ensure function result is at least a list
    if isinstance(result, np.ndarray) or not isinstance(result, (list, tuple)):
        result = [result]
    else:
        if type(result) != str:
            result = list(result)

    # Convert each item to a 1D numpy array
    for index, item in enumerate(result):

        if type(item) == str or type(item) == dict:
            continue # Skip strings
        try:
            result[index] = np.atleast_1d(np.asarray(item))
        except ValueError:
            if should_serialize(item):
                result[index] = [i.tolist() for i in item]

    return result
```

function_registry.py

```python
# shared_memory_processor/function_registry.py
import inspect, types, importlib.util, os
from shared_memory_processor.decorators import classproperty

class Functions:
    """
    Registry class for managing functions.

    Functions can be dynamically added and later retrieved as a dictionary.
    """

    IGNORE_LIST = ['get_function_list', 'add_function', 'njit','least_squares','wraps']

    @classmethod
    def get_function_list(cls):
        """
      Retrieve all registered functions excluding those in the ignore list.

      Returns:
         dict: Mapping of function names to function objects.
      """
        functions = {}
        # Use dir() to get all attributes; descriptors are automatically called
        for name in dir(cls):
            if name in cls.IGNORE_LIST:
                continue # Skip ignored names
            attr = getattr(cls, name)

            # Check if attribute is a function or method
            if isinstance(attr, (types.FunctionType, types.MethodType)):
                functions[name] = attr
        return functions

    @classmethod
    def add_function(cls, name, func, method_type='instance'):
        """
        Add a new function to the Functions registry.
        """
        if method_type == 'classmethod':
            func = classmethod(func)
        elif method_type == 'staticmethod':
            func = staticmethod(func)

        # Dynamically attach the function to the class
        setattr(cls, name, func)

def import_functions(custom_dir, target_class, method_type='staticmethod'):
    """
    Import functions from Python files in a given directory and register them.
    """
    for filename in os.listdir(custom_dir):
        if filename.endswith('.py') and not filename.startswith('_'):
            module_name = filename[:-3] # Remove '.py' extension
            module_path = os.path.join(custom_dir, filename)
            spec = importlib.util.spec_from_file_location(module_name, module_path)
            module = importlib.util.module_from_spec(spec)
            spec.loader.exec_module(module)

            # Register each function found in the module
            for name, func in inspect.getmembers(module, predicate=inspect.isfunction):
                target_class.add_function(name, func, method_type=method_type)
```

processor.py

```python
# shared_memory_processor/processor.py
import sys, os, json, uuid, random
from datetime import datetime
import numpy as np
from .utils import get_script_dir, should_serialize

current_dir = get_script_dir()
shared_memory_dir = os.path.join(os.path.dirname(current_dir), "shared_memory-main", "python")
sys.path.insert(0, shared_memory_dir)

# Import shared memory operations
from shared_memory import set_shared_memory_path, get_shared_memory_data, set_shared_memory_data

class SHMEM_Processor:
    """
   Processor for managing shared memory segments.

   Attributes:
       BASE_DIR (str): The base directory of this module.
       BUFFER_PATH (str): Directory path for IPC buffer.
       METADATA_PATH (str): File path for shared memory metadata.
   """
    # Set the working directory relative to this file.
    BASE_DIR = os.path.dirname(os.path.abspath(__file__))
    BUFFER_PATH = os.path.join(BASE_DIR, "__ipc_buffer__")
    METADATA_PATH = os.path.join(BUFFER_PATH, "_metadata.json")

    def __init__(self):
        """
        Initialize the processor with a unique session ID and clear existing variables.
        """
        self._session_id = str(uuid.uuid4())
        self.clear_vars()
        # DEBUGGING: change to False to retain input data
        self.OVERWRITE = True

    def clear_vars(self):
        """
        Clear existing shared memory variables by deleting all files in the buffer directory.
        Resets metadata to include current session ID and an empty segments list.
        """
        # Create buffer directory if it doesn't exist
        if not os.path.exists(self.BUFFER_PATH):
            os.makedirs(self.BUFFER_PATH)

        # Remove all files and subdirectories within the buffer
        for file in os.listdir(self.BUFFER_PATH):
            file_path = os.path.join(self.BUFFER_PATH, file)
            if os.path.isfile(file_path) or os.path.islink(file_path):
                os.remove(file_path) # Remove file or link
            elif os.path.isdir(file_path):
                import shutil
                shutil.rmtree(file_path) # Remove directory

        # Reset metadata file with a new session ID and empty segments
        with open(self.METADATA_PATH, 'w') as f:
            json.dump({"session_id": self._session_id, "segments": []}, f)

    def _read_shared_memory_metadata(self):
        """
        Read metadata from the shared memory metadata file and load associated data segments.
        """
        with open(self.METADATA_PATH, 'r') as f:
            metadata = json.load(f)
        segments = metadata.get('segments', [])
        args = []

        # Loop through segments and load shared memory data for valid segment files
        for seg in segments:

            if seg.get("inline", False):
                # Inline data stored directly in metadata.
                data = seg.get("inlineData")
                # If data was serialized as JSON, decode it.
                if seg.get("dataType") == "json":
                    data = json.loads(data)
                args.append(data)
                continue

            file_path = seg.get("filePath")
            if not file_path.endswith(".shmem"):
                continue # Skip non-shared memory files
            set_shared_memory_path(file_path)
            data = get_shared_memory_data()
            if seg.get("dataType") == "json":
                # Deserialize complex data.
                data = json.loads(data)
            else:
                if type(data) != str:
                    data = data.squeeze()
            args.append(data)
        return args

    def _create_shared_memory_segments(self, *args):
        """
        Overwrite existing metadata with new shared-memory segments for each argument.
        """
        buffer_path = self.BUFFER_PATH
        meta_path = self.METADATA_PATH
        OVERWRITE = self.OVERWRITE
        if not os.path.exists(buffer_path):
            os.makedirs(buffer_path)

        # Load existing metadata if it exists
        if os.path.exists(meta_path):
            with open(meta_path, 'r') as f:
                existing_metadata = json.load(f)
        else:
            existing_metadata = {'session_id': self._session_id, 'segments': []}
        session_id = existing_metadata.get('session_id', self._session_id)
        existing_segments = {seg['varId']: seg for seg in existing_metadata.get('segments', [])}
        if OVERWRITE:
            existing_segments_overwrite = {
                seg['varId']: seg
                for seg in existing_metadata.get('segments', [])
                if not seg.get('inline', False)
            }
            existing_segments = {}

        # Create a new shared memory segment for each argument
        for value in args:
            var_id = '{:08x}'.format(random.getrandbits(32))

            if (isinstance(value, np.ndarray) and value.size == 1) or isinstance(value, np.generic):
                value = value.item()

            if isinstance(value, (int, float, str, bool, bytes)):
                segment_entry = {
                    'varId': var_id,
                    'source': 'Python',
                    'inline': True,
                    'inlineData': value,
                    'dataType': type(value).__name__,
                    'dimensions': [],
                    'timestamp': datetime.now().strftime('%Y-%m-%dT%H:%M:%S')
                }
                existing_segments[var_id] = segment_entry
                continue # Skip creating a shared memory file.

            if OVERWRITE:
                if existing_segments_overwrite:
                    var_id = existing_segments_overwrite.popitem()[0]

            segment_file = os.path.join(buffer_path, var_id + ".shmem")
            set_shared_memory_path(segment_file)

            if should_serialize(value):
                # For non-uniform data types, use JSON serialization.
               data_type = "json"
               dimensions = len(value)
               value = json.dumps(value)

            else:
                # Determine dimensions and data type based on value type
                if isinstance(value, np.ndarray):
                    dimensions = list(value.shape)
                    data_type = str(value.dtype)
                else:
                    try:
                        dimensions = [len(value)]
                    except Exception:
                        dimensions = []
                    data_type = type(value).__name__

            set_shared_memory_data(value)

            # Add segment info to existing segments dictionary
            existing_segments[var_id] = {
                'varId': var_id,
                'source': 'Python',
                'filePath': segment_file,
                'dataType': data_type,
                'dimensions': dimensions,
                'timestamp': datetime.now().strftime('%Y-%m-%dT%H:%M:%S')
            }

        # Update metadata with new segments
        updated_metadata = {
            'session_id': session_id,
            'segments': list(existing_segments.values())
        }

        # Write updated metadata back to file
        with open(meta_path, 'w') as f:
            json.dump(updated_metadata, f, indent=4)
        return updated_metadata

    def call(self, func):
        """
        Call a function wrapped with shared memory synchronization using this processor.

        Args:
            func (callable): Function to be synchronized and executed.
        """
        # Import the sync decorator locally to avoid circular imports
        from shared_memory_processor.decorators import sync_shared_memory
        decorated = sync_shared_memory(self, func)
        decorated()
```