**User Manual**

**Contents**

[Introduction 1](#_Toc210916570)

[GUI Overview 4](#_Toc210916571)

[Mode Selection Section 5](#_Toc210916572)

[Input Section – Algorithm Selection 5](#_Toc210916573)

[Function Section – Main Function and Parameters 5](#_Toc210916574)

[Function Selection 6](#_Toc210916575)

[Function Parameters Sub-Section 6](#_Toc210916576)

[User-Defined Parameters 7](#_Toc210916577)

[Execution Settings Section 7](#_Toc210916578)

[Execution Log Section 8](#_Toc210916579)

# Introduction

Computational complexity refers to the quantitative description of the resources required by an algorithm to solve a computational problem. Typical resources include execution time, memory usage, number of processor instructions, and any other measurable quantity needed to complete the task. This complexity is usually expressed as a function of the input size , where the output is the amount of the selected resource needed to process an input of size .

There are generally two ways to determine computational complexity: a **theoretical approach**, which derives expressions from the algorithm’s structure and mathematical properties, and an **empirical approach**, which measures the actual resource consumption during execution. For some algorithms, obtaining a theoretical expression is straightforward; however, for complex problems with multiple nested operations, varying data sizes, and nontrivial memory-access patterns, the theoretical derivation becomes cumbersome or impractical. In such cases, an empirical strategy offers a direct and reproducible means of characterizing performance.

In this work we present an automated framework for experimental evaluation of computational complexity. The program has as input a dataset composed of different input size folders; each folder could contain multiple samples of identical size. The program iterates through the folders, processes each sample using the algorithm under test, and records performance metrics using the Linux perf utility. These metrics include the number of executed CPU instructions, processor cycles, and wall-clock execution time.

Once all samples have been processed, the program applies statistical regression to model the relationship between input size and each measured resource. The resulting regression curves provide an **experimental profile of the algorithm’s computational complexity**, enabling quantitative comparisons and evidence-based performance assessment without requiring a theoretical derivation.

**Installation**

The computational performance evaluation framework operates by quantifying the hardware-level behavior of algorithms through measurements of CPU instructions, processor cycles, and total execution time using the *perf* utility, a native Linux tool that provides direct access to processor performance counters. Since *perf* requires low-level kernel access, it operates exclusively under Linux environments. For systems based on Microsoft Windows, this requirement is satisfied through the **Windows Subsystem for Linux (WSL)**, which provides a fully functional Linux kernel capable of running *perf* natively.

The installation process begins with the activation of WSL. This step is carried out from the Windows PowerShell terminal with administrative privileges by executing the command:

wsl –install

This command enables the necessary system features, installs the default Linux distribution (typically Ubuntu), and sets WSL2 as the active version. Once the installation is completed, a system restart is required. After the reboot, the Linux environment can be accessed either from the Start menu or by typing wsl in the PowerShell or Command Prompt terminal. During the first execution, it is necessary to define a username and password to finalize the initialization process of the Linux subsystem. After this initial configuration, the Linux terminal is ready for software installation.

The automated setup of the environment is performed entirely within WSL using a Python-based installation script named setup.py. Before executing the script, it is essential that the file be accessible from within the WSL environment. If the script was created or stored in the Windows filesystem, it must be copied into the WSL home directory. This can be accomplished by navigating to the corresponding path under /mnt/, where all Windows drives are automatically mounted. For example, if the script is located on the Windows desktop, the following command copies it to the WSL home directory (it’s necessary to change the <username> with the corresponding username):

cp /mnt/c/Users/<username>/Desktop/setup.py ~/

Once the script has been transferred, it can be executed directly from the Linux terminal. In most modern versions of Ubuntu distributed through WSL, Python 3 is preinstalled by default. If the interpreter is not present, it can be installed manually by executing:

sudo apt update && sudo apt install -y python3

With Python available, the automated setup is initiated using the command:

python3 setup.py

The setup.py script was developed to perform the entire configuration process autonomously and reproducibly. It begins by verifying that the execution environment corresponds to a WSL instance through inspection of the system descriptor /proc/version. Once verified, it updates the package repositories, installs all development tools, and retrieves the Microsoft WSL2 kernel source tree from the official GitHub repository. It then installs the necessary libraries for the compilation of the *perf* utility, such as those related to event tracing, compression, stack unwinding, and numerical computations. The compilation is carried out using internal kernel implementations of the libraries libtraceevent and libtracefs to ensure compatibility with the Microsoft-maintained kernel version. After successful compilation, the *perf* binary is automatically copied into /usr/local/bin/, making it globally accessible.

The script integrates an internal logging system that records every operation in real time. All progress information and potential errors are simultaneously displayed on screen and stored in a log file named perf\_setup.log, located in the home directory. This record provides a complete trace of the installation and facilitates verification of each step. Additionally, the script supports optional execution parameters to modify its behavior: the argument --skip-update omits the initial system update when it is already current, while --only-build-perf restricts the process to recompiling *perf* without reinstalling dependencies.

Upon completion, the script verifies the correct installation of *perf* by executing perf --version and performing a test measurement with sudo perf stat -- sleep 0.1. The resulting output contains a summary of processor performance events, confirming that the profiler operates correctly within the system. A final report is displayed summarizing the total execution time and indicating the location of the log file.

This procedure ensures a deterministic and reproducible configuration of the Linux environment, eliminating manual dependency management and potential inconsistencies between kernel and library versions. As a result, the system obtains a fully functional *perf* utility compiled directly from the Microsoft kernel sources, providing accurate and stable hardware-level measurements for computational performance evaluation.

# GUI Overview

This section provides a complete description of the graphical user interface (GUI) of the *Performance Analysis Tool*. The GUI has been designed to guide the user step-by-step through the process of selecting, configuring, and executing computational performance evaluations of algorithms.

The interface is divided into **five main sections** (see Figure 1), each one corresponding to a specific stage in the workflow — from selecting the operation mode to viewing the execution results.

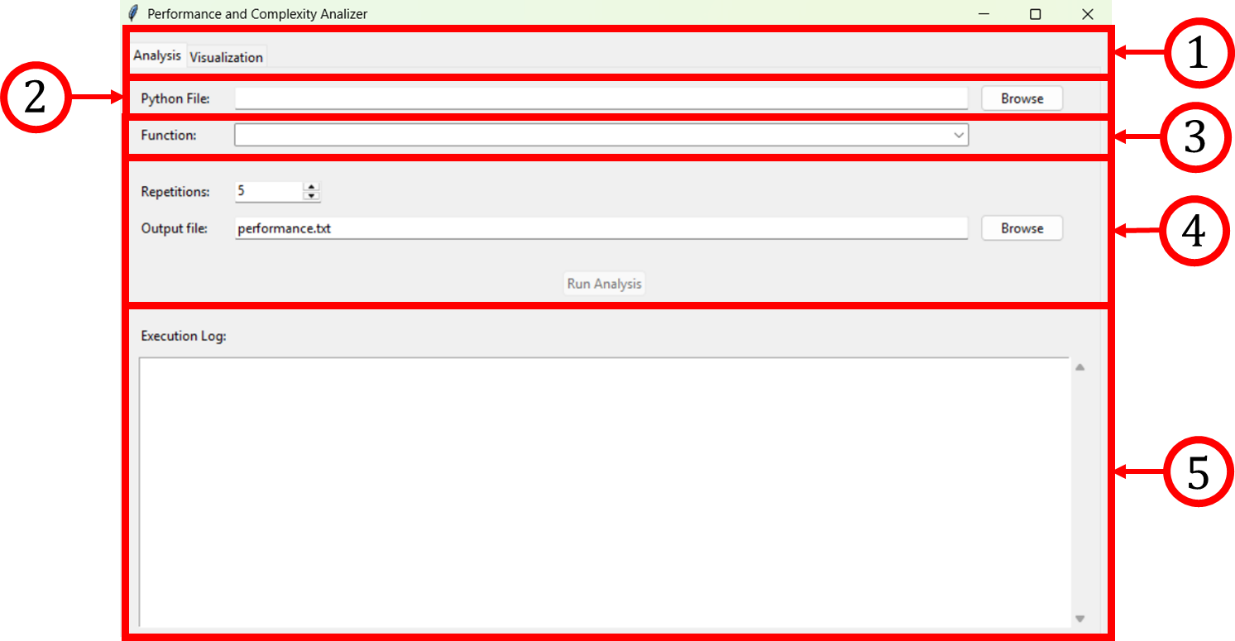
****

Figure 1. GUI sections

## Mode Selection Section

This is the **first section** located at the top of the GUI.  
Here, the user must select **the operation mode** in which the system will run. The available options are:

* **Analysis Mode:**  
  When this mode is selected, the system will execute the algorithm and perform a computational complexity analysis using the Linux perf utility. This mode is intended for users who wish to evaluate how efficiently an algorithm performs in terms of execution time, CPU cycles, and instructions executed.
* **Visualization Mode:**  
  This mode is used only to visualize the results of a previously executed analysis. No computation is performed in this mode. Instead, the system loads the stored performance data and presents it graphically or in textual form.

## Input Section – Algorithm Selection

The **second section** allows the user to select the algorithm to be analyzed.

1. Click the **“Browse”** button.  
   This action opens a file selection window.
2. Navigate to the location of the Python source file (.py) that contains the algorithm.
3. Select the file and click **“Open.”**

After selecting the file, the system automatically scans the file and extracts all **function definitions** contained in it. Only valid Python files (.py) containing at least one function definition (def) will be accepted.

## Function Section – Main Function and Parameters

Once the algorithm file has been selected, the **Function Section** becomes active.  
This section contains two major components.

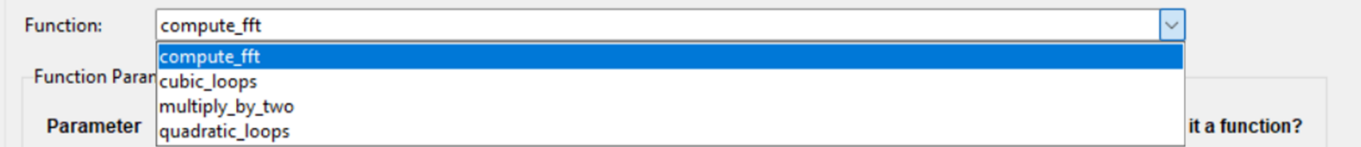
****

Figure 2. Function menu example.

### Function Selection

A dropdown menu labeled **“Function:”** appears in this area (See figure 2).  
The menu lists all the functions that the system automatically detected in the selected source file.

* The user must select from this list the **main function** that represents the algorithm entry point.
* Once the user selects the main function, the system dynamically generates a new sub-area in the interface called **“Function Parameters.”**

### Function Parameters Sub-Section

This sub-area displays all parameters of the selected function (see Figure 3).  
Each parameter is represented as a labeled input field where the user must provide the corresponding value.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 3. Function example parameters.

**Default Parameter**

The **first parameter** displayed is a default system parameter used to define the **set of input images** that the algorithm will process.

* The user must click **in the folder icon** to select a folder.
* Inside that folder, there must be multiple **subfolders (or just one, not recommended)**, each one containing images of different resolutions or sizes (for example: 64x64, 128x128, 256x256, etc.).
* The algorithm will process each folder sequentially during the performance analysis.

### User-Defined Parameters

For each parameter:

* The user must enter the desired value in the corresponding text field.
* To the right of each field, there is a small **checkbox labeled “Is it a function?”**

If a parameter corresponds to a **function** (that is, if the selected algorithm expects another function as an argument), the user must **check this box**.

**Example:**  
If your main function is defined as:

def test(a, b, function):

...

then the parameter function must be marked as **“Is it a function?”**, so the system treats it correctly during execution. Failing to mark this box for function-type parameters may cause execution errors or incorrect performance readings.

## Execution Settings Section

After defining all the function parameters, the fourth section allows the user to configure the execution parameters for the analysis itself. Here, the user must specify:

1. Number of Iterations:  
   Defines how many times the algorithm will process the entire set of images, enter a positive integer (e.g., 5 means the algorithm will run 5 times over each subfolder of images).
2. Output File Name:  
   The user must specify the name of the file where the performance results will be stored.
   * The name must end with the extension .txt (e.g., output\_analysis.txt).
   * The system will save this file in the working directory.

Tip:  
Choose a descriptive name (e.g., fft\_analysis\_256.txt) to easily identify which algorithm and dataset were used.

## Execution Log Section

The **final section** is the **Execution Log**, located at the bottom of the interface, this area provides **real-time feedback** on the current state of the system during execution.

The log displays:

* The start and end time of each operation.
* The function being processed.
* The current iteration.
* The detected dataset being analyzed.
* Any warnings or errors encountered during execution.

The log area is continuously updated and allows users to **monitor the entire process** without accessing the terminal.

# Visualization Step

When the Visualization Mode is selected in the main interface, the system activates a specialized section designed for graphical analysis of previously obtained results. This section contains two primary controls: “Load Results File” and “Generate Plots” (see Figure 4).

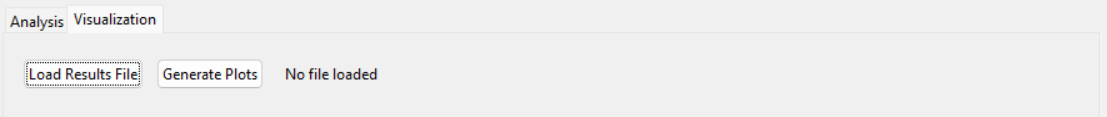


Figure 4. Visualization section.

By pressing the “Load Results File” button, a file selection window appears, allowing the user to navigate through the file system and choose the specific .txt file that contains the computational performance data generated in a previous analysis. Once the file is successfully selected, the system internally parses the recorded data, preparing it for visualization and statistical fitting.

After loading the results file, the user must press the “Generate Plots” button. This action triggers the automatic computation and rendering of two plots that appear directly within the GUI:

1. **Computational Complexity Fit:**

This plot shows the fitted curve that represents the relationship between the number of operations and the input data size. It visually depicts how the number of basic operations scales with the problem dimension, allowing the user to identify whether the algorithm exhibits linear, quadratic, or higher-order growth.

1. **Time Complexity Fit:**

This plot displays the empirical relationship between the total execution time and the input data size. It serves to verify the temporal scaling behavior of the algorithm and provides an independent confirmation of the computational complexity obtained in the previous plot.

Both visualizations are automatically generated based on the loaded dataset, ensuring consistency between the analytical and empirical evaluations. No manual configuration is required beyond selecting the correct results file and initiating the plotting process.